UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

TRABAJO FIN DE GRADO

Tecnología Blockchain en el ámbito de la pobreza energética

Iván Saiz de Castro
Tutor: José Manuel Suárez Fernández
Ponente: Iván González Martínez

JUNIO 2018

TECNOLOGÍA BLOCKCHAIN EN EL ÁMBITO DE LA POBREZA ENERGÉTICA

AUTOR: Iván Saiz de Castro
TUTOR: José Manuel Suárez Fernández

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2018

Resumen

Hasta hace pocos años, cualquier aplicación web era controlada por un único individuo o realización, el cual tiene el poder de todos los datos y manipulación de estos, y el funcionamiento depende de él mismo.

Con la aparición del Blockchain, la manera de gestionar aplicaciones y los datos de estas cambia radicalmente. Blockchain es una base de datos distribuida, que se organiza en bloques en los cuales se almacena un registro criptográfico de todas las operaciones realizadas, que previamente son validadas por una red de nodos independientes repartidos por todo el mundo a través de un algoritmo de consenso. La información almacenada en los bloques no se puede alterar gracias a que los bloques están enlazados mediante un hash, que en resumen son algoritmos unidireccionales que a partir de una entrada consiguen una salida que nunca será igual para distinta entrada. Todos los bloques pueden ser auditables por cualquier miembro de la red de nodos. Esto es una breve explicación, en este trabajo se explicará cómo funciona y en que consiste Blockchain con más detalle.

Existen numerosas plataformas que siguen el modelo de Blockchain pero que además permiten el uso de contratos inteligentes (*Smart Contracts),* que en resumen son contratos programables que implementan reglas de negocio y cuyo código queda registrado en la red y puede ser ejecutado de forma distribuida por los diferentes nodos de la red, esto nos permite desarrollar aplicaciones descentralizadas con lógica basadas en la tecnología Blockchain.

Dada esta tecnología, hay numerosos casos de uso a realizar en los cuales se demanda transparencia, seguridad, inmediatez de las operaciones y fiabilidad.

Para este trabajo de fin de grado se ha elegido un caso de uso que tiene que ver con la pobreza energética actual en España. Según el informe Pobreza, vulnerabilidad y desigualdad energética elaborado por la Asociación de Ciencias Ambientales (ACA), 5,1 millones de personas pasan frío en invierno, 3,2 millones de personas retrasan el pago de sus facturas energéticas y 1,2 millones dedican el 20% de sus ingresos a dichos pagos. Para evitar esto, se ha puesto en marcha el nuevo bono social eléctrico que establece descuentos de hasta el 40% en la factura de la luz para familias en situación de vulnerabilidad severa.

Dado este contexto, se ha desarrollado una aplicación web basada en *Smart Contracts* sobre Ethereum [1] para gestionar este bono social, desde la cual se puede solicitar el bono, gestionar a los solicitantes, pagar facturas, incluso hacer donaciones. Todas las operaciones y datos se almacenan en una Blockchain asegurando así transparencia en la gestión del bono, fiabilidad, inmediatez en pagos para evitar cortes de luz y un proceso auditable con información inmutable.

Palabras clave

Blockchain, Ethereum, Solidity, Contratos Inteligentes, DApp, Pobreza energética, Bono social, Cadena de bloques, Aplicación descentralizada.

Abstract

Until a few years ago, any web application was controlled by a single individual or realization which has the control of all data and manipulation of these, and the operation depends on itself.

With the advent of Blockchain, the management of applications and their data changes radically. Blockchain is a distributed database, which is organized in blocks in which a cryptographic record of all the operations performed is stored, which are previously validated by a network of independent nodes distributed throughout the world through a consensus algorithm. The information stored in the blocks cannot be altered because the blocks are linked by hash function. All blocks can be audited by any member of the node network. This is a brief explanation, in this paper it will be explained how it works and what Blockchain is all about in more detail.

There are numerous platforms that follow the Blockchain model but that also allow the use of Smart Contracts, which in short are programmable contracts that implement business rules and whose code is registered in the network and can be executed in a distributed way by the different nodes of the network, this allows us to develop decentralized applications with logic based on Blockchain technology.

Given this technology, there are numerous cases of use to be made in which transparency, safety, immediacy of operations and reliability are demanded.

For this Bachelor Thesis, a use case has been chosen that is related to current energy poverty in Spain. According to the report Poverty, Vulnerability and Energy Inequality by the Association of Environmental Sciences (ACA), 5.1 million people are cold in winter, 3.2 million people delay paying their energy bills and 1.2 million people spend 20% of their income on these payments. To avoid this, the new electric social voucher has been implemented, which establishes discounts of up to 40% on the electricity bill for families in situations of severe vulnerability.

Given this context, a web application has been developed based on Smart Contracts on Ethereum to manage this social bonus, from which you can request the bonus, manage applicants, pay invoices, even make donations. All operations and data are stored in a Blockchain, ensuring transparency in the management of the bonus, reliability, immediacy of payments to avoid power cuts and an auditable process with immutable information.

Keywords

Blockchain, Ethereum, Solidity, Smart Contracts, DApp, Energy Poverty, Social Bonus, Decentralized Application.

Agradecimientos

En primer lugar quería dar las gracias a todo el equipo docente del grado Ingeniería de Tecnologías y Servicios de Telecomunicación por transmitir sus amplios conocimientos a todos los alumnos, consiguiendo formar a grandes profesionales del área, sin vosotros no serían posibles los grandes avances tecnológicos que estamos viviendo en las últimas décadas y tampoco sería posible que a día de hoy yo hubiese realizado este TFG sin los previos conocimientos que me habéis aportado.

Quería agradecer también a mi tutor José Manuel Suárez, la enorme confianza que ha depositado en mí desde el principio proponiéndome este TFG, ha sido un privilegio desarrollar este proyecto junto a ti, estando siempre en plena disposición de resolver dudas y avanzar conjuntamente consiguiendo todos los objetivos que fijamos al principio. Siempre estaré agradecido por todos los conocimientos que me has transmitido trabajando junto a ti.

De igual forma agradecer también a mi ponente, Iván González Martínez, por apoyarme y ayudarme a realizar este trabajo, desde el principio mostró plena amabilidad y siempre me ha estado asesorando y ayudando a lo largo de todo el TFG.

Por supuesto dar las gracias a mis padres, a mi hermano y a mi pareja, que son los que siempre han estado apoyándome durante toda esta etapa día a día mostrando confianza y estando siempre orgullosos de mi recorrido. Ellos han sido los más importantes durante esta dura pero gratificante etapa y lo seguirán siendo toda la vida.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

1 Introducción

1.1 Motivación

Esta memoria de TFG está motivada por la enorme repercusión que tendrá la tecnología Blockchain en los próximos años. La cadena de bloques rompe con el clásico desarrollo de aplicaciones en las que un miembro o entidad tienen el poder total sobre esta. Blockchain es una base de datos distribuida, que se organiza en bloques en los cuales se almacena un registro criptográfico de todas las operaciones realizadas, que previamente son validadas por una red de nodos independientes repartidos por todo el mundo a través de un algoritmo de consenso. La información almacenada en los bloques no se puede alterar gracias a que los bloques están enlazados mediante una dirección hash. Todos los bloques pueden ser auditables por cualquier miembro de la red de nodos.

Existen numerosas plataformas que siguen el modelo de Blockchain pero que además permiten el uso de contratos inteligentes (*Smart Contracts),* que en resumen son contratos programables que implementan reglas de negocio y cuyo código queda registrado en la red y puede ser ejecutado de forma distribuida por los diferentes nodos de la red, esto nos permite desarrollar aplicaciones descentralizadas con lógica basadas en la tecnología Blockchain. Una aplicación descentralizada o DApp, se define como una aplicación de código abierto que opera de forma autónoma y en la que todos los cambios deben decidirse por consenso de usuarios, además los datos de la aplicación tienen que almacenarse en una cadena de bloques de una red distribuida. Como se verá a lo largo de este TFG, todo esto se puede conseguir gracias a Blockchain y los Smart Contracts.

Tras varios meses de investigación y aprendizaje sobre esta tecnología, empezamos a pensar posibles casos de usos reales para realizar un proyecto piloto con el fin de demostrar la capacidad de desarrollar una aplicación descentralizada y probar las cualidades de la cadena de bloques.

Viendo la aparición del nuevo bono social de luz concedido a personas que cumplen una serie de requisitos y que con ello se encuentran en situación de pobreza energética, y tras la noticia del fallecimiento de una anciana en Reus por un incendio provocado por una vela debido a que tenía el suministro de luz cortado, el tutor de este TFG tuvo la brillante idea de estudiar la manera en la que se concede este bono social y proponerme realizar un proyecto piloto con tecnología Blockchain que permita solicitar y gestionar este bono social dejando grabado en una cadena de bloques todas las gestiones realizadas a través de la plataforma, consiguiendo así transparencia, mayor rapidez en la gestión, y un sistema auditable sin ser vulnerable a modificaciones en los datos. Se decide desarrollar la aplicación con tecnología Ethereum puesto que permite la utilización de Smart Contracts, es una de las plataformas Blockchain más desarrolladas y maduras hasta la fecha y durante los meses de investigación comprobamos que es de las que más documentación tiene y esto nos puede ayudar durante el desarrollo del proyecto.

1.2 Objetivos

El principal objetivo de este TFG es demostrar la capacidad para realizar aplicaciones de uso real sobre tecnología Blockchain en concreto Ethereum, pudiendo así descentralizar numerosas aplicaciones que requieren de transparencia, fiabilidad y de terceros, por ejemplo bancos que podrían ser omitidos disminuyendo así los costes. Además el objetivo más importante es aplicar la base del conocimiento adquirido durante el grado en las distintas asignaturas que estaban orientadas al área de software, como son Programación I, Programación II, Sistemas Distribuidos, Redes, entre otras, además de demostrar la

cualidad que para mí es la más importante que he adquirido durante estos años en el grado, que es la capacidad de aprender rápido adaptándome a nuevas tecnologías y saber solventar con fortaleza los problemas que surgen por el camino.
Para este TFG se va a desarrollar una plataforma que permite solicitar y gestionar a compañías eléctricas un bono social, teniendo un registro único e inmutable de todas las operaciones. Por tanto este trabajo deberá cumplir los siguientes requisitos:

- Desplegar una red privada Ethereum que permita utilizar el proyecto piloto a desarrollar sin los costes que requiere hacerlo en la red pública Ethereum.
- Grabar un registro único de todos los solicitantes del bono social en la cadena de bloques de la red anteriormente desplegada.
- Automatizar la concesión del bono social a través de un Smart Contract.
- Desarrollar una función que permita pagar las facturas de los más vulnerables, dejando registro único de estos pagos.
- Posibilidad de realizar donaciones para este fin, dejando toda la trazabilidad en la cadena de bloques.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos con los cuales se pretende poner en contexto al leyente y tratar de entender el diseño y desarrollo de una aplicación descentralizada sobre una red Blockchain:

- **Capítulo 2: Estado del arte.** En este capítulo se hace una introducción al Bitcoin y a los orígenes de Blockchain, explicando su funcionamiento, la base de esta tecnología y las cualidades de esta. Además se pone en contexto al leyente de la actual situación de pobreza energética en España y se explica en qué consiste el Bono Social.
- **Capítulo 3: Diseño.** Este capítulo incluye el diseño funcional y técnico de la aplicación web, y la arquitectura de una aplicación descentralizada sobre una red Ethereum, en concreto de la plataforma a desarrollar.
- **Capítulo 4: Desarrollo.** En este capítulo se detalla el desarrollo de las principales funciones de los Smart Contracts programados en Solidity necesarios para la plataforma y el desarrollo de la parte "backend" encargada de interactuar con los Smart Contracts y la red Blockchain, haciendo especial énfasis en funciones de web3js que serán utilizadas para desplegar Smart Contracts, e interactuar con estos.
- **Capítulo 5: Integración, pruebas y resultados.** Este capítulo contiene el resultado obtenido a la hora de probar la compilación y despliegue de Smart Contracts en Remix, un IDE (Integrated Development Environment) basado en navegador web que permite escribir contratos inteligentes en Solidity, compilarlos y desplegarlos. Además contiene los resultados del despliegue de Smart Contracts generados desde la plataforma ya integrada al completo con la red privada de Ethereum generada en el **Anexo A**.
- **Capítulo 6: Conclusiones y trabajo futuro.** En este capítulo se definen las conclusiones obtenidas tras el desarrollo de la aplicación descentralizada, así como opciones de trabajo futuro para continuar con este proyecto o implementar nuevos casos de uso con esta tecnología.

2 Estado del arte

2.1 Bitcoin y el origen de Blockchain

2.1.1 La revolución de la cadena de bloques

El origen de la tecnología Blockchain se remonta al 2009, año en el que Satoshi Nakamoto implementa la criptodivisa denominada Bitcoin y establece su base algorítmica basada en técnicas criptográficas. Una criptodivisa se define como una moneda virtual que sirve para intercambiar bienes y servicios a través de un sistema de transacciones electrónicas sin la necesidad de un intermediario.

Satoshi Nakamoto introduce simultáneamente dos conceptos radicalmente novedosos:

- El Bitcoin, una moneda peer-to-peer digital y descentralizada Peer-to-peer significa que existe una conexión directa entre nodos u ordenadores sin necesidad de un servicio intermediario.
- La cadena de bloques (blockchain) basada en la prueba de trabajo.

Con su innovadora cadena de bloques, Satoshi Nakamoto consigue dar solución a la problemática de obtener consenso descentralizado para las monedas digitales.

En los protocolos de moneda digital pre-Bitcoin, se asume que todos los participantes en el sistema son conocidos, definiendo por ejemplo que si hay "N participantes, el sistema puede tolerar hasta N/4 actores maliciosos". Sin embargo, estos protocolos eran vulnerables ante "*sybil attacks*", donde un simple atacante crea miles de nodos simulados para tomar control de la red.

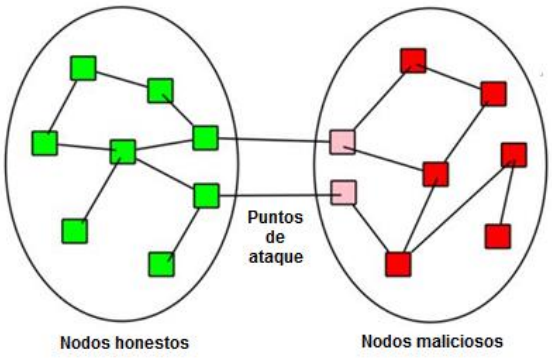


Figura 2-1: Sybil Attacks

La innovación proporcionada por Satoshi Nakamoto combina: un protocolo descentralizado de consenso muy sencillo, basado en nodos, juntando distintas transacciones en un "bloque" cada aproximadamente 10 minutos, creando una cadena de bloques que crece indefinidamente y, el mecanismo de prueba de trabajo, mediante el cual los nodos adquieren el derecho a participar en el sistema y añadir nuevos bloques a la cadena de bloques.

Así surge el blockchain o cadena de bloques, que podemos definirlo como una estructura de datos que implementa una contabilidad distribuida (ledger), entendida como un registro

criptográfico de todas las operaciones realizadas y previamente validadas por una red de nodos independientes a través de un algoritmo de consenso.

El mecanismo de prueba de trabajo se trata de un problema de elevado nivel computacional que tratará de resolver cada uno de los nodos. El primer nodo que encuentre la solución al problema podrá incluir un nuevo bloque en la cadena, y recibirá como compensación un determinado número de Bitcoins. Estos Bitcoins no existen previamente, sino que se crean como parte de este proceso de consenso. Es lo que coloquialmente se denomina "minar Bitcoins". Los nodos con mayor poder de cómputo seguirían teniendo mayor influencia en la red (incorporarían más bloques a la cadena), pero llegar a tener más poder de computación que la combinación de todo el resto de nodos es mucho más complejo que simular miles de nodos.

A pesar de su simplismo, el protocolo se ha mostrado lo suficientemente robusto como para dar inicio a una nueva revolución tecnológica basada en el concepto de cadena de bloques y consenso descentralizado

2.1.2 Protocolos de consenso descentralizado

Al ser Bitcoin un sistema de moneda descentralizada, debe combinarse un sistema para controlar el estado de las transacciones y un sistema de consenso con el fin de garantizar que todo el mundo está de acuerdo en el orden de las operaciones.

El protocolo de consenso descentralizado de Bitcoin requiere que los nodos de la red continuamente intenten producir paquetes de transacciones los anteriormente mencionados bloques. Cada bloque contiene, entre otros elementos:

- Un identificador (*nonce*)
- Una marca temporal (*timestamp*)
- Una referencia (hash) al bloque inmediatamente anterior
- La lista de todas las transacciones que han sido incluidas en el bloque.

Con el paso del tiempo, este proceso genera una cadena de bloques (blockchain) persistente y en constante crecimiento, que se actualiza constantemente para representar el último estado del ledger de Bitcoin

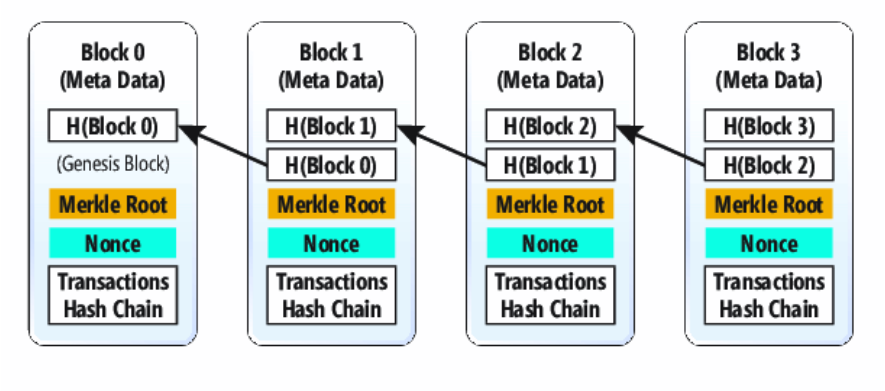


Figura 2-2: Ejemplo cadena de bloques

El protocolo de consenso utilizado por Bitoin se denomina *Proof of Work* (Prueba de trabajo). Los pasos que sigue el protocolo de consenso son: Primero se crea la transacción, por ejemplo Roy quiere enviar a Jen 90BTC (paso 1), a continuación esta transacción es distribuida por todos los nodos de la red (paso 2), para que estos validen las transacciones y creen un nuevo bloque (paso 3), todos los nodos mediante prueba de trabajo (*proof of work)* trataran de resolver un problema de alto coste computacional para decidir quién será el encargado de encadenar el bloque (paso 4), el primero que lo consiga será el encargado de encadenar el bloque y se distribuirá la nueva legder con un bloque más (paso 5).

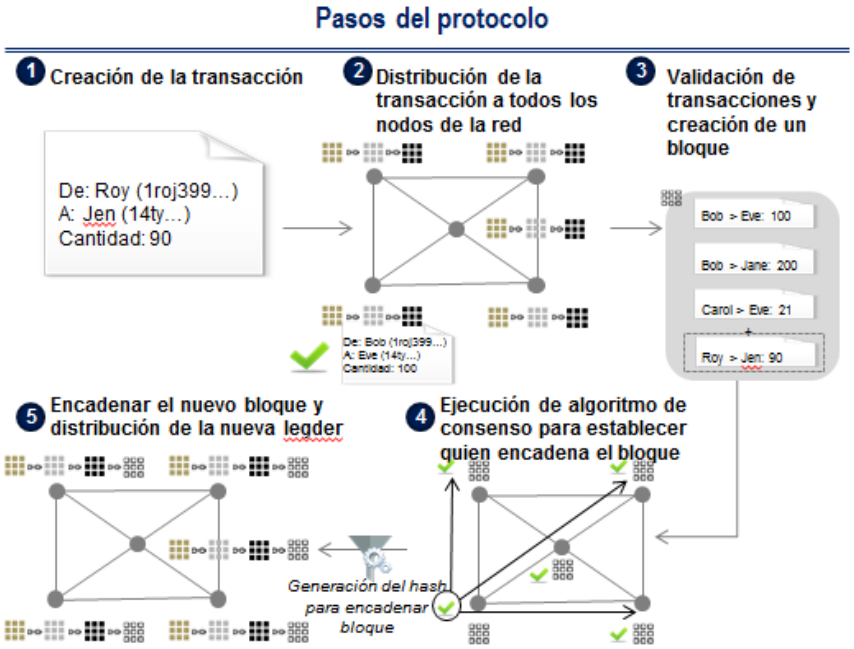


Figura 2-3: Pasos del protocolo de consenso de Bitcoin

Quizás la prueba de trabajo sea el concepto más costoso de comprender de la red Bitcoin, trataré de explicarlo de manera sencilla. Como se ha mencionado anteriormente, cada bloque contiene una serie de datos denominada *nonce*. Los nodos de la red deben encontrar la serie correcta de manera que el bloque completo satisfaga una condición arbitraria, esto requiere calcular múltiples veces la función SHA256 [2] sobre el nuevo bloque que se quiere incluir en la cadena hasta encontrar un resultado que tenga un cierto número de ceros iniciales previamente fijados.

El primer nodo que consiga un resultado válido podrá incluir el bloque en la cadena. Como compensación, recibirá una cantidad prefijada de Bitcoins, así como las comisiones incluidas en las transacciones del bloque, a estas comisiones en el caso por ejemplo de Ethereum, se las denomina *gas*.
Al ser SHA256 una función semialeatoria de "output" totalmente impredecible, la única forma de encontrar una solución válida es por prueba y error, incrementando en cada prueba el "*nonce"* del bloque, el "*nonce"* es un campo de 4 bytes que determina el número de ceros que tiene que contener el hash de un bloque a la hora de calcular la función SHA256.

El objetivo es hacer que la creación de un nuevo bloque sea computacionalmente muy costosa, impidiendo de este modo que un atacante pudiera rehacer la cadena de bloques en su propio interés.
El grado de dificultad del cálculo es recalibrado por la red cada 2016 bloques, de forma que en media se genere un bloque cada 10 minutos.
Si aumenta el poder de computación de los nodos de la red, los bloques se generan en media en menos de 10 minutos, por tanto se aumentará la dificultad de la prueba.
Si disminuye el poder de computación de los nodos de la red, los bloques se generan en media en más de 10 minutos, por tanto se disminuirá la dificultad de la prueba.

Otros protocolos de consenso más innovadores y con menor coste energético y medioambiental son los siguientes:

- ***Proof of Stake*:** A diferencia de *Proof of Work*, el encargado de introducir el siguiente bloque en la cadena no es quién consigue la resolución de un problema de alto coste computacional, sino quien posee mayor número de monedas. Algunos ejemplos que usan *Proof of Stake* son Nxt o Peercoin. Es habitual que *Proof of Stake* se use junto a *Proof of Work* para crear algoritmos más seguros y con menor consumo de recursos. Faircoin o Ethereum son ejemplos del uso de algoritmos híbridos.

- ***Proof of Authority*:** En las redes que utilizan este tipo de algoritmo de consenso, las transacciones y los bloques son validados por nodos definidos como nodos validadores, estos nodos tienen el poder para realizar las transacciones con bajo coste computacional lo que genera que la red sea más rápida que en otro tipo de consensos, por seguridad no permite la generación de dos bloques consecutivos por parte de un mismo nodo.

- ***Proof of Resources:*** Este proceso mide la capacidad para almacenar y recuperar fragmentos de datos. Esto depende de los siguientes criterios de la computadora del minero: velocidad de la CPU, disponibilidad de ancho de banda, espacio del disco y tiempo en línea. Esto permite que la prueba sea una entidad útil, mensurable e inmediatamente verificable. Safecoin utiliza este tipo de algoritmo de consenso.

- ***Proof of Burn*:** Consiste en eliminar monedas para conseguir introducir un nuevo bloque. En realidad las monedas eliminadas no son destruidas sin ningún tipo de beneficio, sino que se invierten en equipos de minería virtual, y cuantas más monedas consigas eliminar más potencia tiene el equipo virtual. En resumen funciona como un proceso de "minería virtual": Cuando eliminas monedas compras un equipo de minería virtual. Cuánto más dinero eliminas, más potente es tu equipo. Cada equipo virtual te da el derecho de minar por un largo período, como los mineros de verdad. Pero gradualmente pierden competitividad, tal como los equipos de minería que quedan obsoletas por el avance tecnológico. Slimcoin utiliza este método.

2.1.3 Ejemplo de ataque malicioso

Dado que las partes del protocolo protegidas criptográficamente son seguras, el atacante tendría como objetivo la única parte no protegida con criptografía: el orden de las transacciones. Veamos un intento de ataque, que sigue la siguiente estrategia:

1- Alice envía 100 BTC a un comerciante (Bob) a cambio de algún producto.
2- Esperar a la entrega del producto.
3- Alice genera otra transacción enviando los mismos 100 BTC a sí mismo.
4- Alice trata de convencer a la red que la transacción enviada a si mismo fue la que se realizó primero.

Siguiendo los pasos de la estrategia, en el momento en que Alice envía 100BTC a Bob se genera una transacción, y unos minutos después esta transacción queda grabada en un bloque, por ejemplo el 1.000. Como cada bloque nuevo apunta al bloque anterior, es decir, el 1.001 apunta al 1.000, el 1.002 al 1.001 y así sucesivamente, todos confirmarán la transferencia de Alice a Bob, Bob esperará un número de confirmaciones (nuevos bloques) que considere seguro, momento en el cual acepta el pago y Bob entrega o envía el producto a Alice.

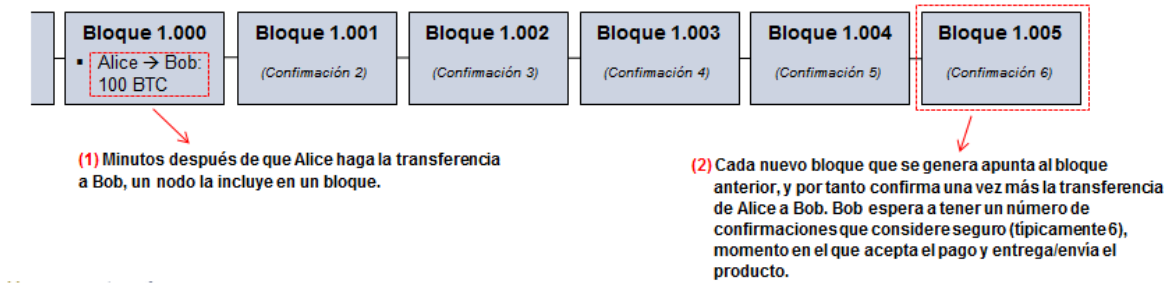


Figura 2-4: Ejemplo ataque malicioso (1)

Una vez entregado el producto, Alice generará una nueva transacción enviando los mismos 100BTC a sí mismo, creando una cadena de bloques paralela partiendo del nuevo bloque 1.000. Lo que evitará que Alice cumpla su objetivo malicioso será que el nuevo bloque creado tiene distinto hash al contener distinta información, por tanto el bloque 1.001 no apuntará al bloque malicioso 1.000.

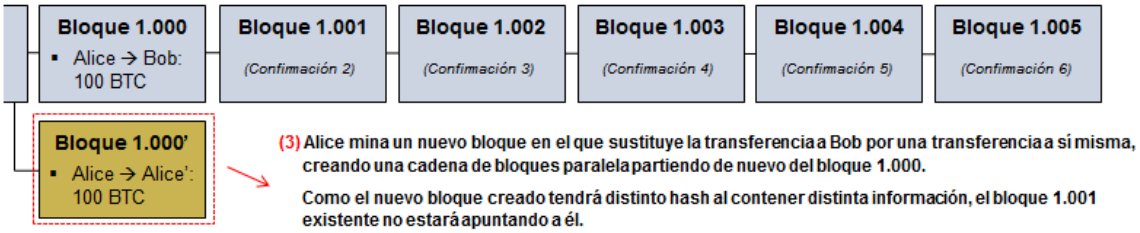


Figura 2-5: Ejemplo ataque malicioso (2)

Ninguno de los bloques ya existentes del 1.001 al 1.005 se encadenarán con el bloque malicioso de Alice, por tanto se producirá una bifurcación de la cadena, y en estos caso la regla es que la cadena más larga, es decir, la que está respaldada por mayor prueba de trabajo se toma como verdadera y los miembros legítimos seguirán trabajando sobre ella.

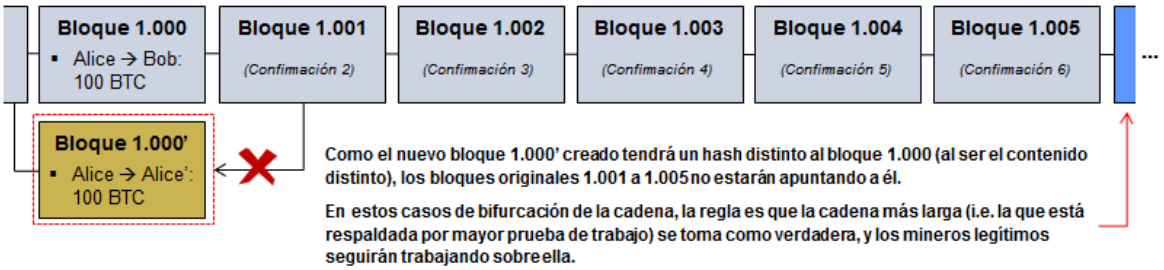


Figura 2-6: Ejemplo ataque malicioso (3)

La única forma de que Alice consiga que su cadena sea más larga y así que los nuevos bloques se enganchen a su cadena y conseguir que su ataque sea efectivo, es produciendo nuevos bloques a una velocidad superior que el resto de los mineros de la cadena legítima, para ello Alice debería tener más poder computacional que todo el resto de mineros en conjunto (51% Attack).

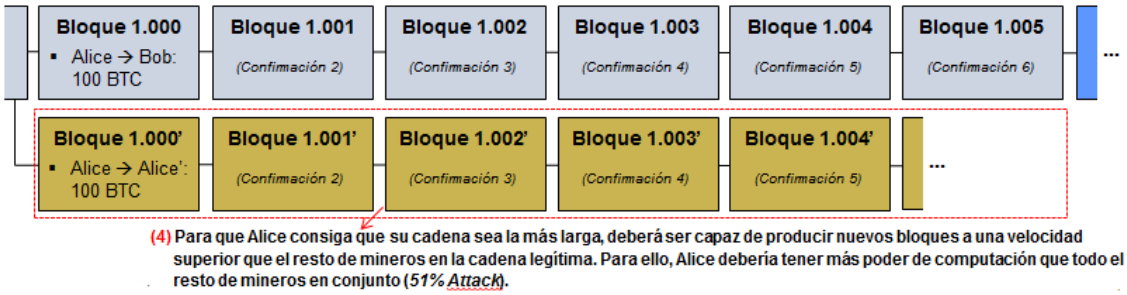


Figura 2-7: Ejemplo ataque malicioso (4)

Con este pequeño ejemplo, se puede ver como no es sencillo manipular la cadena de bloques ya que conseguir tener el 51% de poder computacional es muy complejo o casi imposible.

2.2 Tipos de Blockchain

Dentro de esta tecnología podemos diferenciar diferentes tipos de Blockchain según el tipo de acceso.

2.2.1 Blockchains públicas

Los protocolos de Blockchain públicos basados en algoritmos de consenso *Proof of Work (POW)* son de código abierto y no están permisionados, es decir, cualquiera en el mundo puede leer, enviar transacciones y esperar verlas incluidas si son válidas, y cualquier persona puede participar en el algoritmo de consenso.
Como sustituto de la confianza centralizada, las cadenas de bloques públicas están protegidas por criptoeconomía: la combinación de incentivos económicos y verificación criptográfica utilizando mecanismos como es la prueba de trabajo o prueba de participación, siguiendo el principio general en el que cuanto más poder computacional, lo que equivale a más influencia en el proceso de consenso, más cantidad de recursos económicos. Estas cadenas de bloques generalmente se consideran "totalmente descentralizadas". Las Blockchain públicas más conocidas son Ethereum y Bitcoin.

2.2.2 Blockchains privadas

Una Blockchain totalmente privada es una cadena de bloques donde los permisos de escritura se mantienen centralizados en una organización, el acceso a la red es permisionado, es decir, para realizar transacciones un organismo central te tiene que dar permiso. Los permisos de lectura pueden ser públicos o restringidos en una extensión arbitraria. Generalmente estas Blockchains no requieren de costes económicos ya que el valor del *gas* al realizar una transacción puedes establecerlo como nulo.
Para la realización de este trabajo se ha escogido este tipo de Blockchain ya que al tratarse de un proyecto piloto queríamos que el coste en las transacciones fuese nulo para poder realizar numerosas pruebas sin ningún coste.
Algunas de las más famosas son Hyperledger (de la Fundación Linux), R3 (un consorcio de bancos internacionales para desarrollar soluciones bancarias de blockchain privada) o Ripple (un protocolo para facilitar las transferencias internacionales de dinero).

2.2.3 Blockchains híbridas

Las Blockchain híbridas son una combinación de las públicas y privadas. En una Blockchain híbrida los nodos participantes son invitados, pero todas las transacciones son públicas. Es una cadena de bloques donde el proceso de consenso se controla mediante un conjunto de nodos preseleccionados; por ejemplo, uno podría imaginar un consorcio de 15 instituciones financieras, cada una de las cuales opera un nodo y de las cuales 10 deben firmar cada bloque para que el bloque sea válido.

El derecho a leer la Blockchain puede ser público o restringido a los participantes, y también hay rutas híbridas como el hash raíz de los bloques que son públicos junto con una API que permite a los miembros del público hacer un número limitado de consultas y obtener pruebas criptográficas de algunas partes del estado de la Blockchain. Estas cadenas de bloques se pueden considerar "parcialmente descentralizadas".

En este trabajo se propondrá como trabajo futuro integrar este caso de uso en *Alastria,* una red Blockchain híbrida/semipública, independiente, permisionada y neutral. Ya que en el caso de redes híbridas puede no existir costes económicos como es en el caso de *Alastria* en el que el valor del *gas* al realizar transacciones es nulo.

2.3 Características del Blockchain

2.3.1 ¿Qué aporta?

Permite el registro de cualquier activo digitalizable, sea este criptomonedas, instrumentos financieros o “Smart Contracts” (contratos programables que implementan reglas de negocio y cuyo código queda registrado en la cadena de bloques y puede ser ejecutado de forma distribuida por los diferentes nodos de la red).

Inmediatez de las operaciones ya que quedan confirmadas en el momento en el que se registran en la cadena.

Seguridad y privacidad a través de reglas criptográficas que permiten el registro inviolable de las operaciones realizadas y una autenticación segura de los partícipes.

Trasparencia ya que todas las operaciones se encuentran registradas en la cadena de bloques y pueden ser auditadas por cualquier miembro de la red.

Eliminación de un punto único de fallo ya que si un nodo se ve comprometido el resto de la red permanece seguro auditando la cadena de bloques.

Disminución de costes ya que se elimina la intermediación (operativa "peer to peer") realizada por un tercero para validar y registrar las operaciones.

2.3.2 Eficiencia

Blockchain resuelve problemas de costes, duplicación y reconciliación. La mayor reducción se vería reflejada en los gastos, especialmente en transacciones internacionales, donde Blockchain ofrecería una infraestructura más rápida y barata.
La posibilidad de compartir un mismo "*ledger"* por varios bancos permitiría eliminar información duplicada de partidas/contrapartidas entre bancos, agilizando el proceso de reconciliación de transacciones financieras y permitiendo ahorros en sistemas y equipos actualmente utilizados para realizar este proceso.
En nuestro caso de uso del bono social que veremos más adelante, la posibilidad de que todos los solicitantes queden registrados en la misma cadena de bloques existiendo un registro único por solicitante, y todas las compañías eléctricas y administración puedan acceder a estos datos agilizaría el proceso de gestión y de auditoría, así como los pagos.

Al realizarse pagos directamente entre comprador y vendedor, se reduciría, por una parte, el rol de los bancos, cámaras de compensación y bancos centrales como intermediarios, y por otra, los tiempos asociados a las transacciones, principalmente internacionales.

Mediante el uso de Blockchain, es posible la creación de Contratos Inteligentes, es decir, programas que realicen transacciones de forma automática al cumplirse ciertos criterios.
Los contratos inteligentes son algoritmos informáticos que facilitan, verifican y hacen cumplir la negociación sin necesidad de tener una cláusula contractual.
Una cláusula de un Contrato Inteligente es ejecutada por un ordenador a partir de una condición pre-programada. De esta forma, se obtiene un servicio de depósito de garantía en tiempo real con un costo de operación cercano a cero.
Los beneficios de los contratos inteligentes incluyen menores costes de contratación/ ejecución y cumplimiento, recomendables para transacciones de bajo valor. Más adelante se explica con detalle en qué consiste un Contrato Inteligente (*Smart Contract).*

2.3.3 Transparencia

Blockchain contiene información sobre todas las transacciones que se han realizado, incluyendo el concepto y la hora. Contiene la historia completa de todas las transacciones que se han ejecutado en la red con una marca temporal. Cualquier persona en cualquier momento puede acceder a consultar ese registro sabiendo que esa información está actualizada y es consistente con el resto de nodos de la red. Mantiene la traza de las transacciones de manera continua, el emisor y el receptor de cada transacción son registrados, y cada una de las transacciones está disponible públicamente para inspección.

Dicha información es inmutable, no puede ser eliminada ni modificada, y de acceso público, lo cual genera enorme confianza por parte de los usuarios.
Blockchain ofrece un alto nivel de transparencia gracias a que muchos participantes pueden acceder a la cadena de bloques, tener una copia, y comprobar cada registro. Esta

característica puede ser empleada para demostrar ante un Regulador que la base de datos no ha sido editada ni modificada para fines fraudulentos. Esta solución incrementa la confianza depositada por parte de inversores de productos complejos u opacos.

2.3.4 Robustez

El sistema de "ledgers" distribuidos implica que la información no está en un solo computador, sino que existen múltiples copias de la misma, asegurando la disponibilidad e integridad de la información.
Blockchain es una base de datos con información horaria estampada e inmutable de cada transacción, que se replica en servidores de todo el mundo. Al no tener un punto central para poder dirigir los ataques la red se vuelve mucho más segura, cualquier agente maligno precisaría un enorme esfuerzo para conseguir realizar un ataque al sistema, ya que debería atacar a todas las copias de forma simultánea.
El sistema de verificación de las transacciones es muy robusto, todas las transacciones tienen que ser aprobadas por múltiples nodos del proceso antes de poder ser incorporadas a la cadena, asegurando también que la información no pueda ser reescrita, como ya hemos visto en el ejemplo de ataque.
Cada transacción es verificada por una comunidad de usuarios autorizados, en lugar de por una autoridad central, lo que le otorga una gran resistencia a alteraciones, además es automáticamente administrada, manteniendo un histórico que dificulta su reversión.
La tecnología ofrecida en Blockchain es, por tanto, resistente a cambios no autorizados o falsificaciones, ya que los participantes de la red tienen información inmediata de cada uno de los cambios ocurridos dentro del "ledger".

2.3.5 Seguridad

Blockchain está basado en un algoritmo criptográfico, que garantiza seguridad no sólo al usuario individual sino también al sistema, haciéndolo, si no infranqueable, altamente difícil de atacar.
La seguridad y precisión de la información es mantenida gracias a algoritmos criptográficos, que se encargan de autenticar y administrar el contenido y las transacciones, asegurando que cada una de las copias del ledger coincide exactamente con el resto (Reconciliation Through Cryptography).
El acceso a los "ledgers" se hace a través de claves publicas/privadas, los registros son añadidos con una única clave criptográfica que garantiza que el participante correcto ha añadido el correcto registro siguiendo las correctas reglas.
No sólo es seguro a nivel de sistema sino también para los usuarios individuales, quienes para mover el dinero necesitan una clave privada asociada a la clave pública.
El hecho de tener un "ledger" distribuido y replicado en millones de ordenadores representa una posibilidad de ataque ínfima, mejorando la seguridad para los bancos que la utilicen.

2.3.6 Privacidad

No hay una relación directa entre ambas partes que figuran en el intercambio, lo cual garantiza la privacidad de los usuarios que, sin embargo, no tienen un perfecto o completo anonimato gracias a la traza e información que queda registrada en el sistema.

Los usuarios pueden hacer transacciones de su dinero de manera privada, sin intermediarios, lo que reduce el tiempo y costes de las transacciones.

Pese a ofrecer privacidad el sistema Blockchain permite obtener la traza e identificar el usuario que realizó los cambios en el sistema. Por tanto, los usuarios no tienen garantizado el anonimato. Por ejemplo, al transformar bitcoins en monedas reales habrá que pasar la regulación de blanqueo de capitales y financiación de terrorismo.

2.4 Contratos Inteligentes (Smart Contracts)

2.4.1 ¿Qué son?

Se definen como contratos escritos como programas informáticos, sustituyendo el lenguaje legal de un documento impreso. Sobre él, pueden definirse reglas y consecuencias del mismo modo que en un documento legal tradicional, pero el *Smart Contract* también puede tomar información como input, procesarla según las reglas definidas y adoptar las medidas que se requieran como resultado.
Son scripts modulares, repetibles y autónomos que normalmente se ejecutan en una Blockchain. El contrato queda almacenado en una dirección específica dentro de la Blockchain, y al producirse un evento contemplado en el contrato, se envía una transacción a esa dirección. La máquina virtual distribuida ejecuta las cláusulas del contrato usando los datos recibidos en la transacción.
Un *Smart Contract* tiene validez, sin depender de autoridades, debido a su naturaleza: es un código visible por todos y que no se puede cambiar al existir sobre la tecnología Blockchain, la cual le da ese carácter descentralizado, inmutable y transparente.

2.4.2 ¿Qué aportan?

Automatización de las operaciones, ya que las cláusulas se ejecutan sin necesidad de intervención humana tras su registro en el ledger.
Disminución de costes ya que se elimina la intermediación (operativa P2P) realizada por un tercero para configurar el contrato, registrarlo y ejecutarlo en su caso.
Seguridad y privacidad a través de reglas criptográficas que permiten el registro inviolable de los contratos.
Alto nivel de customización, pudiendo incluirse en el código casi cualquier lógica empresarial basada en datos que se quiera (valoración del contrato, elaboración de métricas de riesgo, generación de órdenes de cobro/pago automáticas, conexión a fuentes certificadas para verificar el cumplimiento de las cláusulas del contrato, etc.).

2.5 Pobreza energética

Según los datos del Banco Mundial, 1.200 millones de personas en el mundo (el 17% de la población), no tienen acceso a la electricidad, mientras que 2.700 millones (un 38% de la población), ni siquiera disponen de condiciones adecuadas de cocina.

En Europa, según Eurostat, el 9,4% de la población no puede mantener una temperatura adecuada en su hogar en invierno.

En concreto, en España, según el informe Pobreza, vulnerabilidad y desigualdad energética elaborado por la Asociación de Ciencias Ambientales (ACA), 5,1 millones de personas pasan frío en invierno, 3,2 millones de personas retrasan el pago de sus facturas energéticas y 1,2 millones dedican el 20% de sus ingresos a dichos pagos.

En este sentido, en algunos países de la UE se contaba ya con una definición establecida de "pobreza energética", a los cuales se ha unido España recientemente con la aprobación del RD 897/2017 por el que se regula la figura del consumidor vulnerable, el bono social y otras medidas de protección para los consumidores domésticos de energía eléctrica.

La reciente puesta en marcha del nuevo bono social eléctrico establece descuentos de hasta el 40% en la factura de la luz para familias en situación de vulnerabilidad severa.

Además, la nueva regulación obliga a las empresas comercializadoras a la financiación del bono social como un servicio de interés general y regula un mecanismo para evitar los cortes de suministro en los casos con mayor riesgo.

2.5.1 Bono Social

¿Qué es?

El bono social es un mecanismo de descuento en la factura eléctrica, fijado por el Gobierno, con el fin de proteger a determinados colectivos de consumidores económica o socialmente más vulnerables. El bono social es un descuento que se aplica sobre el PVPC (precio voluntario para el pequeño consumidor), a un límite máximo de energía en el periodo de facturación.

Condiciones y criterios socio-económicos:

Puede solicitar el Bono Social todo sujeto considerado "consumidor vulnerable", "consumidor vulnerable severo" o "consumidor vulnerable severo en riesgo de exclusión social" según los criterios legales, siempre que reúna las siguientes condiciones:

Condiciones generales:

- Que el titular sea persona física.
- Que el Punto de Suministro para el que se solicite la aplicación del bono social sea el de la vivienda habitual.
- Que el titular está acogido al PVPC.
- Que la potencia contratada para dicho punto de suministro sea igual o inferior a 10 kW.

Condiciones particulares:

1.-Consmidor vulnerable (25% de descuento):

- Renta anual de la unidad familiar inferior o igual a determinados coeficientes sobre el IPREM (con excepciones en caso de discapacidad, violencia de género o terrorismo).
- Familias numerosas.
- Pensionistas del Sistema de Seguridad Social (jubilación o incapacidad permanente).

2.-Consmidor vulnerable severo (40% de descuento):

- Renta anual de la unidad familiar inferior al 50% de los coeficientes fijados para el consumidor vulnerable.
- Familia numerosa con renta inferior a IPREM x2.
- Pensionistas del Sistema de Seguridad Social (jubilación o incapacidad permanente) con renga igual o inferior al IPREM.

3.-Consumidor en riesgo de exclusión social (50% de descuento):

- Consumidores vulnerables severos que, además son atendidos por los servicios sociales de una Administración autonómica o local.

Procedimiento de tramitación y gestión de solicitudes:

Aquellos consumidores que consideren que reúnen los requisitos para acceder al bono social deben presentar la correspondiente solicitud (a través un modelo estándar) y la documentación acreditativa de sus circunstancias socio-económicas:

- NIF o NIE del titular del punto de suministro y, en su caso, de todos los miembros de la unidad familiar.
- Certificado/s de empadronamiento.
- Libro de familia.
- Título de familia numerosa (en su caso).
- Certificados de la Seguridad Social y otros organismos competentes para la acreditación de circunstancias especiales (en su caso).

En concreto, las solicitudes del Bono Social se gestionarán con la compañía comercializadora de referencia por alguna de las siguientes vías de comunicación:

- Por teléfono (número disponible en la página web de la comercializadora de referencia).
- En las oficinas de la empresa.
- Por fax.
- Correo postal.
- Página web.

Una vez recibida la solicitud y documentación completa, el comercializador dispone de 15 días hábiles, para comunicar al solicitante el resultado de las comprobaciones efectuadas (éstas se llevarán a cabo a través de la plataforma informática disponible a tal efecto en la Sede Electrónica del Ministerio de Energía, Turismo y Agenda Digital).

La empresa comercializadora de referencia podrá firmar convenios de colaboración con las distintas Administraciones autonómicas o locales competentes. Asimismo, la Administración autonómica o local cuyos servicios sociales estén atendiendo al consumidor que cumpla los requisitos para ser considerado vulnerable severo, podrá comunicar este hecho a la comercializadora de referencia.
Por otro lado, cuando las Administraciones autonómicas o locales hayan creado y puesto en marcha un registro administrativo de puntos de suministro de electricidad para los consumidores en riesgo de exclusión social, podrán solicitar la colaboración de la Administración General del Estado para compartir los datos, de tal forma que los comercializadores de referencia puedan efectuar las consultas correspondientes en el mismo.

Procedimiento de suspensión en los contratos de suministro de electricidad:

El procedimiento de suspensión para aquellos contratos acogidos al PVPC y para los contratos en mercado libre correspondientes al suministro de electricidad de personas

físicas en su vivienda habitual con potencia contratada igual o inferior a 10 kW según la nueva normativa es el siguiente:

- Vencimiento del período de pago desde la emisión de la factura (20 días naturales o, en su caso, en lo establecido entre las partes en contratos de mercado libre).
- Comunicación de la circunstancia al consumidor en el plazo de 2 meses desde la emisión de la factura o en el momento en que se produzca el rechazo del pago si fuera con posterioridad a dicho plazo.
- Debe realizarse por cualquier medio que permita tener constancia de su recepción.
- Se considerará fehaciente cuando se realice por correo certificado, burofax o mediante firma electrónica.
- En caso de notificación infructuosa, se remitirá un segundo requerimiento 7 días hábiles después. Si realizado este segundo requerimiento no es posible su notificación, se especificarán las circunstancias de ambos intentos y se tendrá por efectuado el trámite.
- Los consumidores sin bono social dispondrán de 2 meses para realizar el pago, mientras que los consumidores vulnerables dispondrán de hasta 4 meses a partir de la notificación del requerimiento.
- Remisión semanal por medios electrónicos al órgano que designe cada Comunidad Autónoma del listado de puntos de suministro a los que se haya requerido el pago, indicando la fecha a partir de la cual el suministro puede ser suspendido.
- El fin de esta comunicación es poner en conocimiento de las Administraciones Autonómicas estas situaciones de impago y puedan ser adoptadas las medidas necesarias que se consideren oportunas.
- Se requerirá con una antelación de quince días hábiles a la finalización del plazo establecido, comunicando la fecha concreta a partir de la cual el suministro podrá ser suspendido.
- Finalizado el plazo, se podrá solicitar a la distribuidora a través del procedimiento y por los sistemas y medios telemáticos aprobados por la CNMC la suspensión del suministro.

3 Diseño

3.1 Arquitectura de una DApp sobre Ethereum

Gracias a los Contratos Inteligentes se pueden desarrollar numerosas aplicaciones descentralizadas denominadas "DApp" ya que estos nos permiten desarrollar programas informáticos dentro de una Blockchain. Las "DApps" normalmente cuentan con una parte centralizada (aplicación cliente) y otra parte descentralizada que se ejecuta sobre una Blockchain. La arquitectura habitual que presentan es la siguiente:

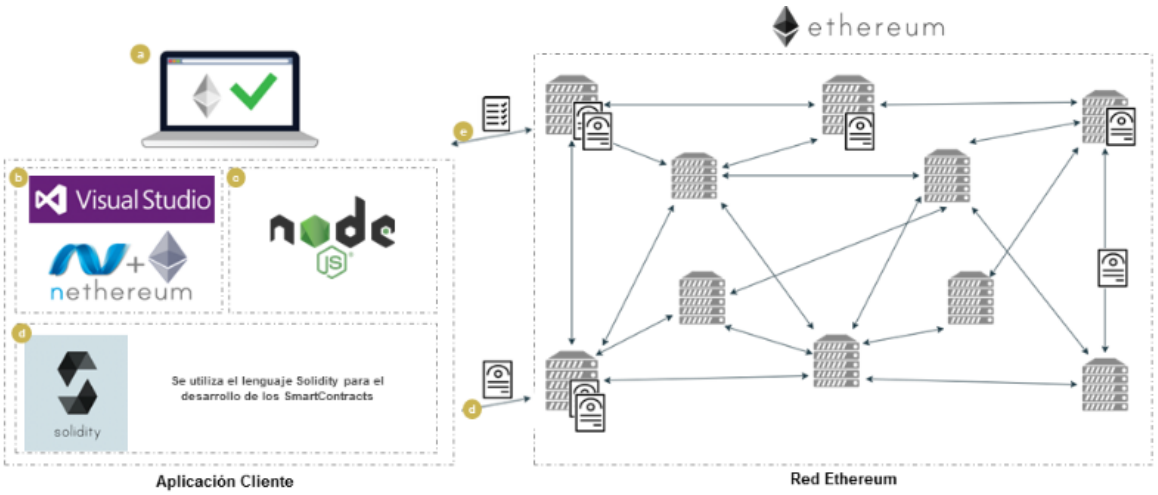


Figura 3-1: Arquitectura de una DApp

Este ejemplo serviría para una DApp sobre Ethereum. Existen varias tecnologías para construir aplicaciones que se comuniquen con una red Blockchain Ethereum. Visual Studio junto con Nethereum nos proporciona un entorno para el desarrollo de aplicaciones basado en C#. Node.js proporciona un entorno para desarrollar aplicaciones basadas en JavaScript. Independientemente de la tecnología utilizada, los Smart Contracta se programan en Solidity. Éstos se despliegan en la red desde una aplicación cliente (ej. Ethereum Wallet), y una vez desplegados son accesibles desde cualquier nodo de la red.
La comunicación entre la parte cliente de la aplicación y los Smart Contracts se realiza mediante transacciones, que llevan asociadas un coste según su complejidad (*gas*), la API utilizada para esta comunicación es web3.

Para este caso de uso se va seguir la arquitectura de la figura 3-1, utilizando para la aplicación web node.js junto con Express esto configura un entorno para el desarrollo de aplicaciones en JavaScript (creamos la parte "Backend" en JavaScript). El motivo de utilización de esta tecnología es que la API de comunicación con nuestro Blockchain Ethereum (web3js) está desarrollada en ese lenguaje.

3.2 Diseño y arquitectura de la plataforma

La plataforma se dividirá en cuatro bloques, clientes/solicitantes, compañías eléctricas, ayuntamientos y ONGS. Para este proyecto piloto los cuatro bloques se encontraran en una misma aplicación web, pero la idea como trabajo futuro es diferenciar los bloques en aplicaciones web distintas.

Todos los bloques compartirán un mismo "home" donde se mostrará en qué consiste el bono social y se tendrá la posibilidad de realizar donaciones a cualquier ONG dada de alta en la plataforma para este fin. En el **Anexo B** se pueden ver todas las imágenes del diseño de las pantallas.

- **Bloque clientes/solicitantes:** Este bloque tendrá una pantalla sencilla en la que el solicitante deberá rellenar todos los datos necesarios para hacer un estudio de si le corresponde el bono social y que tipo de consumidor es: vulnerable, vulnerable severo o en riesgo de exclusión social, que como se ha visto anteriormente esto conllevará a un descuento en la factura del consumo eléctrico.
  Los datos que se pedirán por pantalla son los siguientes: Nombre, Apellidos, Renta, Número de hijos, DNI, Atendido por servicios sociales (SI/NO), Dirección, País, Provincia, Código Postal y Compañía de la que es cliente el solicitante. Además habrá un apartado para adjuntar los documentos necesarios en formato pdf. Al tratarse de un proyecto piloto no implementado en ninguna compañía eléctrica real, se solicitará el importe de la factura para poder hacer simulaciones de pagos, en un caso real se integraría esta plataforma con la base de datos de la compañía para tener acceso a dicha información.
  Todos los datos introducidos serán los parámetros de entrada de un nuevo Smart Contract que se desplegará en la red cada vez que un usuario solicite el bono social, tendrá por nombre *Cliente.sol*. Parte del desarrollo de este Smart Contract se puede ver en el apartado **4.3 Desarrollo de Smart Contracts.**

  Este Smart Contract tendrá la capacidad de almacenar los datos recibidos y determinar el tipo de consumidor que es el solicitante y si está en disposición de obtener el bono social. Para la verificación de la renta se debería desarrollar un *oráculo* que se comunicase con el MINETAD a través de una API, esto no se desarrollará en este trabajo de fin de grado puesto que para un proyecto piloto no es fácil que el MINETAD te facilite este servicio. Para hacer uso de un *oráculo* se desarrollará uno muy sencillo que con el parámetro provincia se comunique con una API de información meteorológica e introducirá en el Smart Contract *Cliente.sol* el clima actual en dicha provincia y asignará una prioridad al solicitante, esto servirá de ayuda para los ayuntamientos y ONGS a la hora de tomar decisiones de a quién pagar la factura en caso de no disponer de fondos suficientes para abordar todos los pagos.
  El lector se podría preguntar, ¿por qué un Smart Contract por cada solicitante y no uno único que almacene la información de todos los solicitantes?, bien, la razón por la que he elegido un Smart Contract por solicitante es la siguiente: imaginemos que cuatro millones de clientes solicitan el bono social a través de la plataforma, un único Smart Contract debería almacenar toda la información de los cuatro millones de solicitantes y cada vez que se produce un cambio en cualquiera de estos o se incluye un nuevo solicitante cambia el estado del contrato, esto requiere del proceso de minado de nuevo para almacenar en la cadena de bloques estos cambios. Parece lógico que minar un contrato con información de cuatro millones de personas cada vez que se produce un mínimo cambio es mucho más costoso que minar un sólo Smart Contract con la información de un único solicitante.
  Otra duda que podría surgir es, ¿Cómo accedes a cada Smart Contract para conseguir información o modificar información si son millones y están desplegados por la red cada uno con su dirección? Pues bien, para simplificar las búsquedas, se va a desarrollar un Smart Contract "maestro" que tendrá únicamente almacenadas

las direcciones de todos los Smart Contract de todos los solicitantes mapeados con su DNI, por tanto simplemente con el DNI podremos obtener la dirección hash del contrato que queremos buscar para acceder a él con esta dirección sin necesidad de realizar búsquedas por toda la cadena de bloques.
Este bloque será público.

- **Boque compañías eléctricas:** En este bloque se dispondrá de diferentes apartados. Tendrá un primer apartado en el cual cualquier compañía eléctrica podrá darse de alta en el sistema, será una pantalla sencilla en la que se tendrán que facilitar los datos de la compañía eléctrica: Nombre Social, CIF y contraseña. Para un proyecto real quizá sean pocos datos, pero como se ha mencionado antes el objetivo de este trabajo de fin de grado no es implantarlo tal como se va a hacer para uso público sino un proyecto piloto para poder demostrar la capacidad de desarrollar una aplicación para un caso real sobre blockchain y mostrar las cualidades de esta tecnología. Una vez dados de alta tendrán otro apartado "Home compañías" en el cual deberán iniciar sesión con su CIF y contraseña y podrán ver todos los solicitantes que tienen en la compañía y el estado de estos, y también podrán ver todos los pagos recibidos de facturas por parte de ONGS y/o ayuntamientos y el balance de su cuenta dentro del sistema.
  De nuevo se desarrollará para este bloque un Smart Contract llamado Company.sol, los parámetros de entrada para el despliegue de este contrato serán nombre, CIF y contraseña, este Smart Contract almacenará la información de la compañía eléctrica, será capaz de recibir pagos y grabarlos en eventos, en la parte de desarrollo veremos en qué consisten los eventos y para qué sirven. Se seguirá la misma estructura que para los solicitantes, un Smart Contract por cada compañía que se dé de alta en el sistema, y se almacenará en el contrato "maestro" la dirección hash del Smart Contract mapeada con el CIF para facilitar las búsquedas.
  A este bloque solamente tendrán acceso las compañías eléctricas.

- **Bloque ayuntamientos:** Este bloque será muy similar al de las compañías eléctricas, tendrá dos apartados también. Un primer apartado en el cual cualquier ayuntamiento podrá darse de alta en el sistema, será una pantalla sencilla en la que se tendrán que facilitar los datos del ayuntamiento: Población, Código Postal y contraseña. Una vez dados de alta tendrán otro apartado en el cual podrán ver todos los solicitantes que tienen en su población y el estado de estos, pudiendo pagar el importe de la factura de aquellos que se encuentren en disposición de la ayuda del bono social con un solo "click". Y otro apartado en el que podrán ver todos las donaciones recibidas por parte de ONGS, los pagos realizados de factura a las compañías eléctricas y el balance de su cuenta dentro del sistema.
  Pará este bloque se desarrollará un Smart Contract llamado Ayuntamiento.sol, los parámetros de entrada para el despliegue de este contrato serán nombre, C.P y contraseña. Este Smart Contract almacenará la información del ayuntamiento, será capaz de recibir donaciones de ONGS, realizar pagos de facturas a compañías eléctricas y grabarlos en eventos. De nuevo un Smart Contract por cada ayuntamiento que se dé de alta en el sistema y se almacenará en el contrato "maestro" la dirección hash del Smart Contract mapeada con el C.P para facilitar las búsquedas.

- **Bloque ONGS**: De nuevo este bloque seguirá el mismo diseño que los dos anteriores, tendrá un primer apartado en el cual cualquier ONG podrá darse de alta

en el sistema facilitando: Nombre Social, Identificador y contraseña. A diferencia de los anteriores bloques, las ONGS tendrán un apartado en el que podrán ver a todos los solicitantes con independencia de la compañía o ayuntamiento al que pertenezcan, pudiendo pagar el importe de la factura de aquellos que se encuentren en disposición de la ayuda del bono social con un solo "click". Además se les facilitará dos pantallas más en las que podrán ver todos los ayuntamientos y compañías con el número y tipo de solicitantes que tienen para ayudarles en la toma de decisiones a la hora de hacer donaciones o pagar facturas.
En este caso se desarrollará un Smart Contract llamado Ong.sol, los parámetros de entrada para el despliegue de este contrato serán nombre, identificador y contraseña. Este Smart Contract almacenará la información de la ONG, será capaz de recibir donaciones, realizar pagos y grabarlos en eventos. De nuevo un Smart Contract por cada ONG que se dé de alta en el sistema y se almacenará en el contrato "maestro" la dirección hash del Smart Contract mapeada con el identificador para facilitar las búsquedas.

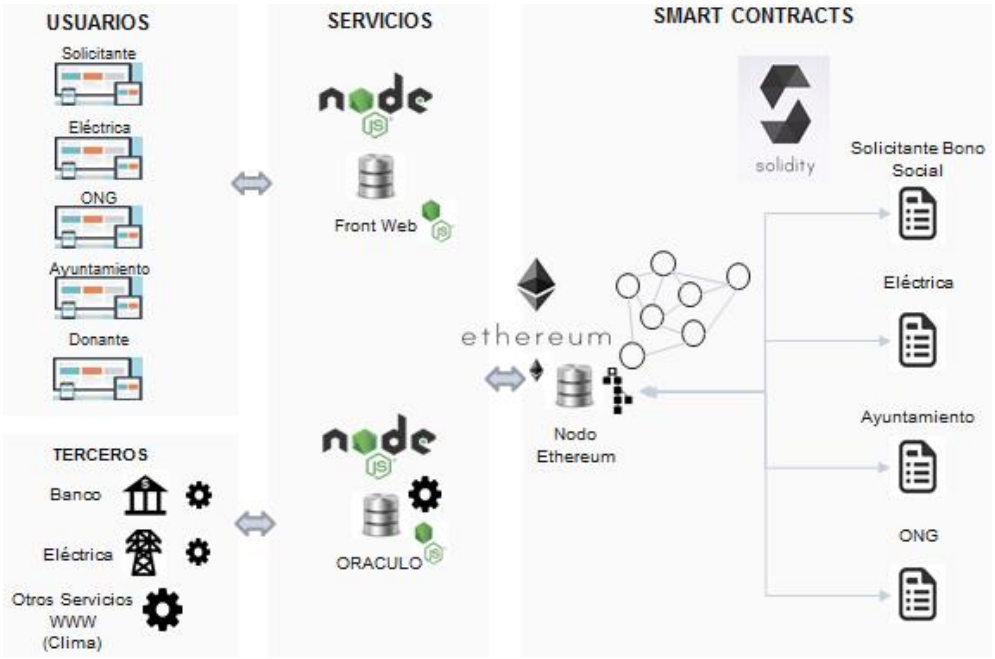


Figura 3-2: Diseño y arquitectura de la plataforma a desarrollar

4 Desarrollo

4.1 Introducción

En este apartado se explica de principio a fin como desarrollar a nivel tecnológico una aplicación descentralizada sobre una red Blockchain. La instalación y despliegue de una red privada Etherum se puede ver detallada en el **Anexo A**. Para este TFG y caso de uso se ha desarrollado sobre una red privada basada en Ethereum, pero el desarrollo sería equivalente si quisiésemos desarrollar una DApp sobre cualquier otra red pública o privada.

4.2 Desarrollo Smart Contracts

Como ya he mencionado anteriormente, los Smart Contracts se programan en Solidity, un lenguaje de programación de alto nivel orientado a contratos. Está influenciado por C ++, Python y JavaScript y está diseñado para trabajar sobre la Máquina Virtual Ethereum (EVM).

La mejor manera de probar los Smart Contracts que se programan en Solidity en este momento es usando Remix. Remix es un IDE basado en navegador web que permite escribir contratos inteligentes en Solidity, compilarlos y ejecutarlos. Yo siempre durante el desarrollo de la plataforma antes de subir los contratos a mi red y desplegarlos, los he probado en Remix para descartar posibles fallos de código. El link de acceso a esta herramienta es: https://remix.ethereum.org, sencillo de utilizar y altamente recomendable para empezar a desarrollar contratos inteligentes.

Como se ha visto en la parte de diseño, son necesarios cinco Smart Contracts: Cliente.sol, Company.sol, Ayuntamiento.sol, Ong.sol y el contrato "maestro" que actuará como base de datos de todas las direcciones hash de los Smart Contracts que lo he llamado ReportCliente.sol. A continuación mostraré un ejemplo de desarrollo de un Smart Contract destacando las funcionalidades más importantes. Para no abrumar al lector con mucho código en este apartado voy a explicar partes del desarrollo de los dos Smart Contracts que se puede decir que son los que más funcionalidad tienen dentro de la plataforma. Para aquel que esté más interesado en leer un poco más acerca de cómo he desarrollado los Smart Contracts puede acceder al **Anexo C** dónde se encuentra parte de código de los demás contratos.

Cliente.sol:

Lo primero de todo es señalar que versión de Solidity estamos utilizando (en mi caso 0.4.0) y darle nombre al contrato que deberá llamarse igual que el archivo .sol. Lo siguiente es definir las variables propias del contrato, variables predefinidas que servirán para comprobar el cumplimiento de distintas "clausulas". En el caso de cliente.sol, las variables que se necesitan para comprobar qué tipo de cliente es el solicitante son: IPREM, número de hijos a partir del cual se considera familia numerosa y potencia máxima contratada para poder optar al bono social.

```
pragma solidity ^0.4.0;

//Contrato de cliente
contract Cliente{

    //variables propias del contrato
    uint min_num_hijos= 3;
    uint IPREM = 7520;
    int potenciaMax = 10;
```

A continuación se definen las estructuras de datos que se van a almacenar en el Smart Contract, es importante señalar que en Solidity una estructura de datos puede contener como máximo 11 variables. Dado que necesito almacenar más de 11 variables, he separado los datos en 3 estructuras: ClientPersonal, ClientHouse y ClientStatus.

```
//Estructura que contiene datos personales, renta, tipo de cliente, y descuento
//que le corresponde
struct ClientPersonal {
    address adr;
    string name;
    string surname;
    uint personal_id;
    uint renta;
    uint num_hijos;
    string typeClient;
    string discount;
    uint prioridad;
    uint idc;
    uint factura;
}

//Estructura que contiene la dirección de la vivienda del cliente, se separa de la
//anterior estructura porque hay limite de variables en las estructuras
struct ClientHouse{
    string street;
    string country;
    string province;
    uint cp;
}

//Estructura que contiene fecha de creacion del contrato del cliente, fecha de impago(en caso de que la
//empresa eléctrica notifique que este cliente no paga), si el cliente es atendido
//por los servicios at_serv_sociales, la potencia contratada por el cliente y la prioridad

struct ClientStatus{
    uint256 creationDate;
    uint256 nonPaymentDate;
    string at_serv_sociales;
    int potenciaContratada;
    uint prioridad;//1 alta, 2 media, 3 baja
}
```

Todo Smart Contract debe tener una función denominada constructor, es la primera función que se ejecuta al crear el contrato, esta función debe llamarse igual que el Smart Contract, en este caso Cliente. A esta función se le pueden pasar parámetros, añadir lógica y llamar a otras funciones si el diseño lo requiere o simplemente una función vacía que lo único que haga sea inicializar el contrato. Como he mencionado anteriormente, cada vez que se solicite el bono social se creará este Smart Contract, por tanto todos los datos que se piden a la hora de solicitarlo serán parámetros de la función constructor Cliente y se almacenarán en las estructuras que he definido para este contrato. Como ya he dicho esta función es la primera en ejecutarse a la hora de desplegar el contrato, entonces lo primero que he hecho ha sido almacenar toda la información, determinar qué tipo de cliente es en función de los parámetros introducidos y asignarle una prioridad.

```
function Cliente(string _name, string _surname, uint _personal_id, uint _renta, uint _num_hijos,
string at_serv_sociales, string _street, string _country, string _province,
uint _cp, uint idc, int _tempmin, uint factura) {

    addClientPersonal(_name, _surname, _personal_id, _renta, _num_hijos, idc, factura);
    addClientHouse(_street, _country, _province, _cp);
    addClientReport(_personal_id, address(this));
    typeConsumer(_renta, _num_hijos, at_serv_sociales, address(this), _personal_id, _cp, idc);
    asignarPrioridad(_tempmin, _renta, _personal_id, address(this));

}
```

Para determinar qué tipo de cliente es y que descuento le pertenece, he desarrollado la función *typeConsumer* que se puede decir que contiene las "clausulas" necesarias para obtener el bono social, el desarrollo de esta función se puede ver en el **Anexo C** también.

En esta sección quiero destacar los eventos. Los eventos permiten el uso de la capacidad de registro del EVM (Ethereum Virtual Machine), que además permite hacer llamadas desde una interfaz de usuario JavaScript para una DApp que escuche estos eventos. Cuando se llama a un evento los argumentos se guardan en el registro de transacciones. Estos registros están asociados con la dirección del contrato y serán incorporados en la blockchain y allí permanecerán siempre. He definido este evento para dejar constancia en la blockchain de a quién se le otorga el bono social (msg.sender), que tipo de descuento y la fecha exacta en la que se inserta estos datos en la blockchain (block.timestamp), después desde node.js podré recuperar el log de eventos con todos los cambios de cada solicitante. La definición de un evento es muy sencilla, queda de la siguiente forma:

```
//Este evento nos permmitirá grabar dirección, estado y fecha cada vez que le invoquemos
//para después poder recuperar un log de eventos desde node js
event status(address adr, string estado, uint256 time);
```

Voy a destacar también la función *addClientReport* ya que contiene una funcionalidad muy útil de solidity que es la comunicación entre distintos Smart Contracts. Esta función lo que hace es añadir en el Smart Contract que antes he llamado "maestro", la dirección del nuevo contrato generado junto con el DNI del solicitante.
Para establecer comunicación entre Smart Contracts, hay que invocar al Smart Contract "llamado" utilizando el mismo nombre con el que se desplegó en la red y haciendo referencia a él con la dirección hash que tiene ese Smart Contract dentro de la blockchain, esta dirección se obtiene cuando despliegas el Smart Contract. Una vez invocado se pueden utilizar todas las funciones que estén definidas en ese contrato. Debo destacar que además de invocarlo es necesario incluir la estructura del Smart Contract llamado, no es necesario incluir toda la lógica, simplemente el constructor y los nombres de las funciones. A continuación muestro como establezco comunicación entre el Smart Contract Cliente y ReportCliente:

```
//Invoco a ReportCliente con la dirección hash correspondiente
ReportCliente report =  ReportCliente (0xB7C08d63D07D05A0e39a60875bf8B475F840902c);

//Utilizo la función aaddClientReport de ReportCliente almacenando dirección y dni
function addClientReport(uint _personal_id, address _adrClient) private{
    report.addClientReport(_personal_id, _adrClient);
}
```

```
//Si queremos comunicarnos con el smartcontract ReportCliente, debemos incluir
//en el mismo documento.sol el smartcontract pero únicamente el
// constructor y la definición de las funciones a las que vamos a hacer referencia,
//NO es necesario incluir todo el código de lógica

//Contrato de reporting
contract ReportCliente{

    uint256 indexClient;


    struct ClientsAddress{
    uint personal_id;
    uint prioridad; //1 Alta, 2 Media, 3 Baja
    uint tipoCliente; //1 exclusión social, 2 vulnerable severo, 3 vulnerable, 4 no cumple requisitos
    address adr;
    address adrCompany;
    }

    mapping (uint => ClientsAddress) public clientAdr;

    address[] numClientes;

    function addClientReport(uint _personal_id, address _adr){

    }

    function setPrioridad(uint _personal_id, uint _prioridad) {

    }

    function setTipoCliente(uint _personal_id, uint _tipoCliente, uint _cp, uint idc)  {

    }

    }
```

Otro elemento que quiero destacar dentro de este SmartContract son los *modifier*. Este elemento de Solidity nos permite limitar el uso de funciones dentro del contrato. Por ejemplo yo los he utilizado para las funciones que permiten cambiar las variables IPREM, potencia máxima permitida y mínimo número de hijos para ser familia numerosa, estas variables pueden cambiar con el tiempo y alguien se tiene que encargar de hacerlo, pero no cualquiera debe cambiarlas a su antojo, por tanto he definido un *modifier* el cual comprueba que quién llama a esta función lo hace con la dirección hash de la compañía eléctrica y así sólo las compañías eléctricas puedan modificar estas variables, lo he llamado *OnlyCompany*. A continuación muestro el código del *modifier* y las funciones que lo utilizan:

```
//Un modifier permite limitar el acceso a una funcion
        modifier onlyCompany{
        require(msg.sender==adrCompany);
        _;
    }
```

```
//la compañia será quien pueda modificar que potencia tiene el cliente contratada
    function setPotenciaContratada(int _potencia, address _adr) onlyCompany{

    clientDataStatus[_adr].potenciaContratada = _potencia;
    status(msg.sender, "Cambio de variable potenciaContratada", block.timestamp );
    }
//funcion que modifica el IPREM, ya que puede variar
    function setIPREM(uint _IPREM)  onlyCompany{

    IPREM= _IPREM;
    status(msg.sender, "Cambio de variable IPREM", block.timestamp );

    }
//funcion para modificar el minimo numero de hijos para ser familia numerosa
    function setMinNumHijos(uint _min_num_hijos) onlyCompany{

    min_num_hijos= _min_num_hijos;
    status(msg.sender, "Cambio de variable min_num_hijos", block.timestamp );

    }
```

ReportCliente.sol:

Este Smart Contract juega un papel importantísimo en la plataforma, actúa como base de datos de todas las direcciones hash de todos los Smart Contracts que se despliegan en el sistema, ya sean clientes, ayuntamientos, ongs o compañías.
Primero de todo como siempre indico la versión y doy nombre al contrato. A continuación he definido las estructuras para clientes, ongs, ayuntamientos y compañías. En el caso de clientes para conseguir información rápida de los datos más importantes almaceno dni, prioridad, tipo de cliente, dirección de su Smart Contract y dirección del Smart Contract de la compañía a la que pertenece. Para ongs simplemente almaceno identificador y dirección. Para los ayuntamientos almaceno código postal, dirección y número de los diferentes tipos de cliente, podría almacenar solo código postal y dirección pero el número de diferentes tipos de cliente se consultará constantemente y si tenemos la información en este contrato la consulta será mucho más rápida. De igual forma pasa con las compañías, almaceno nombre, dirección, identificador y número de diferentes tipos de cliente.

```
pragma solidity ^0.4.0;
//Contrato de reporting
contract ReportCliente{

    struct ClientsAddress{
    uint personal_id;
    uint prioridad;
    uint tipoCliente;
    address adr;
    address adrCompany;
    }

    struct ONG{
    uint idONG;
    address adrONG;
    }

    struct Ayunta{
    uint cp;
    address adrAyuntamiento;
    uint numExclusionSocial;
    uint numVulnerable;
    uint numSevero;
    uint numNoCumple;
    }

struct Comp {
string nameCompany;
address adrCompany;
uint idCompany;
uint numExclusionSocial;
uint numVulnerable;
uint numSevero;
uint numNoCumple;
}
```

Para realizar las búsquedas en este Smart Contract, he "mapeado" las cuatro estructuras. La de los clientes con el DNI, la de las compañías con el CIF, los ayuntamientos con el C.P y las ONGS con el identificador. Esto lo he hecho con *mapping*, básicamente funciona como un diccionario, es como una matriz en la que el índice es el *_KeyType* y los elementos de la matriz son el *_ValueType.* Tiene la siguiente estructura: *mapping(_KeyType => _ValueType) _type name*

En mi caso todos los *_KeyType* son tipo *uint* y los *_ValueType* son las distintas estructuras.

```
mapping (uint => ClientsAddress) public clientAdr;
mapping (uint => ONG) private ONGadr;
mapping (uint => Ayunta) private ayuntaadr;
mapping (uint => Comp) private companyAdrs;
```

Por tanto si queremos obtener por ejemplo a la dirección de un ayuntamiento y el número de distintos tipo de solicitantes que tiene introduciendo sólo el código postal, la función queda de la siguiente forma:

```
function getAddressAyuntamiento(uint _cp)
returns (address adr, uint exclusion, uint severo, uint vulnerable) {
    return(ayuntaadr[_cp].adrAyuntamiento, ayuntaadr[_cp].numExclusionSocial,
    ayuntaadr[_cp].numSevero, ayuntaadr[_cp].numVulnerable);
}
```

4.3 Desarrollo Backend Node.js

En este apartado detallaré las principales funciones que he desarrollado para establecer comunicación entre el Front y los Smart Contracts, no mostraré todo el código ya que es bastante extenso y muchas funciones siguen la misma estructura.

Para el desarrollo de esta plataforma he seguido el criterio de MVC (modelo, vista, controlador). La parte de lógica de comunicación con los Smart Contracts a través de web3 la he desarrollado en la parte "controlador", que es la que voy a abarcar en este apartado y la realmente importante en este trabajo de fin de grado, ya que al tratarse de una DApp no he querido utilizar bases de datos convencionales y por tanto la parte de "modelo" que normalmente es la parte en la que se desarrolla la lógica para almacenar los datos en una base de datos queda obsoleta. La parte de "vista" la he desarrollado en *.hbs* pero no toma relevancia para este trabajo.

Como ya he dicho antes, la API utilizada para interactuar con cuentas y SmartContracts de una red Ethereum es Web3 que trabaja con peticiones http. Además de Web3 y http se necesita el módulo fs (File System) que proporciona una API para interactuar con el sistema de archivos, esto me servirá para poder leer el código de los SmartContracts a la hora de compilarlos, y el módulo solc que es un compilador de Solidity que será útil para compilar los contratos. Por tanto lo primero hay que instanciar al proveedor de Web3 que en mi caso es el nodo que he desplegado anteriormente e instanciar los módulos fs y solc.

```
var http = require("http");
var Web3 = require('web3');
const fs = require('fs');
const solc = require('solc');
var web3 = new Web3("http://localhost:8545");//nodo 1
```

Web3 tiene el objeto eth que es el utilizado para interactuar con cadenas de bloques de Ethereum. Esta API está diseñada para trabajar sobre un nodo local RPC y todas las peticiones http son síncronas, esto quiere decir que estas peticiones bloquean la ejecución del código mientras se procesa la solicitud. Para realizar una petición asíncrona, en la mayoría de las funciones se puede pasar como último parámetro una función "callback" que recoge el resultado o error en caso de que se produzca algún fallo.

Web3.js tiene numerosas funciones con gran potencial para interactuar tanto con cuentas Ethereum como con Smart Contracts, se puede ver toda la documentación de web3.js en: https://web3js.readthedocs.io/en/1.0/web3.html.

La primera función que he desarrollado para empezar a interactuar con la red Ethereum que he montado en el punto 4.2 y ver el funcionamiento, es una función sencilla que recoge todas las cuentas creadas en la red junto con el balance de estas. Si accedemos a la documentación anteriormente indicada, podemos ver como hay dos funciones de web3.eth ya definidas que consiguen esto. Una es web3.eth.getAccounts y la otra web3.eth.getBalance. En el caso de getBalance es necesario indicar la dirección de la cuenta de la cual quieres obtener el balance. La función escrita en node.js queda de la siguiente forma:

```
function listarCuentasBalances(){

  web3.eth.getAccounts(function(error, result){
    if(!error){
      result.forEach(function (address) {
        //console.log(address);
          web3.eth.getBalance(address, function (error, result) {
            if (!error) {
              console.log("address: "+ address+ " balance: "+ result);
            } else {
              console.error(error);
            }
          })
      });
    } else
        console.error(error);
   });

}
```

Viendo que todo funciona correctamente, el siguiente paso que he dado ha sido desarrollar funciones realmente importantes para la plataforma. La primera función útil es la de desplegar el contrato "maestro" en la red ya que este se encargará de almacenar todas la direcciones de todos los SmartContracts que se vayan desplegando en la red, ya sea de clientes, compañías, ONGS o ayuntamientos.

Para compilar y desplegar un SmartContract, realizar cualquier tipo de función que requiera un cambio de estado en el contrato o para realizar cualquier tipo de transacción es necesario tener una cuenta Ethereum con Ether disponible y que esté desbloqueada, ya que como sabemos estas operaciones tendrán un coste de "gas" que será pagado con esta cuenta. Para desbloquear una cuenta si accedemos de nuevo a la documentación de web3, podemos ver que dentro del objeto "personal" hay una función definida para esto, denominada "unlockAccount", a esta función se le pasa como parámetros la dirección de la cuenta y la contraseña que pusimos a la hora de crear esta.

```
ethereumController.unlockAccount = function (address,password,callback){
//0x0b3ecA6DFFE4fFF33dAdC10DAE4cF8c4DE489857"
//password
web3.eth.personal.unlockAccount(address,
 password, 300, function (error, result) {

            if (!error) {
              return callback(address);
            }
});
}
```

Una vez tengo disponible una cuenta dentro de la red, puedo desplegar y compilar mi primer SmartContract ReportCliente.sol. Una vez más accediendo a la documentación dentro del módulo web3.eth vemos que hay un método denominado "Contract" que genera una nueva instancia de un SmartContrac con todos los métodos y eventos definidos dentro de este a través del objeto de interfaz json del SmartContract, normalmente denominado código *ABI o interfaz binaria*. Para generar el *ABI* primero hay que leer el *documento.sol* donde se encuentra el desarrollo del SmartContract, compilarlo con *solc*, generar el *bytecode* y el *ABI* de la interfaz con json. Con esta isntancia vemos que hay un método denominado "deploy", este método se encarga de desplegar el contrato en la cadena de bloques, como argumentos se le pasa el *bytecode* y los parámetros del constructor del SmartContract en caso de que requiera de estos. Dentro de "deploy" la función "send" será

la encargada de realizar la transacción para desplegar realmente el contrato, a esta función hay que indicarle que cuenta será la responsable de cubrir los gastos y cantidad y precio de *gas* dispuestos a pagar. Esta función devuelve el hash de la transacción (con esta dirección hash luego seremos capaces de localizar quién la realizó, cuándo y en que bloque se añadió), como ya hemos visto anteriormente esto tomará unos segundos hasta que la transacción sea incluida en un bloque. Una vez incluida la transacción en la cadena de bloques, devuelve la dirección del SmartContract. A continuación muestro la estructura final de la función que se encarga de desplegar el SmartContract ReportCliente.sol, para los demás contratos las funciones que he desarrollado siguen la misma estructura y por tanto las omitiré en la memoria.

```
ethereumController.compileDeployContract = function(callback){
const input = fs.readFileSync('solidity/ReportCliente.sol');
const output = solc.compile(input.toString(), 1);
const bytecode = output.contracts[':ReportCliente'].bytecode;
const abi = JSON.parse(output.contracts[':ReportCliente'].interface);
//console.log(abi);
//console.log(bytecode);


console.log("\ncompilando contrato...");
var contract = new web3.eth.Contract(abi);

console.log("\ndesplegando contrato...");
contract.deploy({
    data: '0x'+bytecode
  })
.send({
    from: '0x0b3ecA6DFFE4fFF33dAdC10DAE4cF8c4DE489857',
    gas: 3000000,
    gasPrice: '30000000000'
}, function(error, transactionHash){

})
.on('error', function(error){
  console.log(error);
})
.on('transactionHash', function(transactionHash){
  console.log("\ntransactionHash:" + transactionHash);
console.log("\nRecibiendo dirección contrato...");
})
.on('receipt', function(receipt){
return callback(receipt.contractAddress, abi);
})
.on('confirmation', function(confirmationNumber, receipt){
 })
.then(function(newContractInstance){
});

}
```

Junto con esta función y las tres siguientes que voy a contar como las he desarrollado, que son: recoger eventos de un SmartContract, extraer datos de un SmartContract, y modificar/añadir datos a un SmartContract; considero que es suficiente para poder realizar cualquier tipo de lógica y aplicación que se nos ocurra interactuando con SmartContracts de una red Blockchain.

Para recoger los eventos que he desarrollado en los Smart Contracts en el apartado 4.3, primero hay que hacer referencia al Smart Contract que ha generado el evento de nuevo llamando al método “Contract” pero esta vez además de pasar por parámetro el *abi*, hay que pasar también la dirección del Smart Contract en el que queremos buscar los eventos. Si accedemos una vez más a la documentación de web3, en el módulo “Contract” vemos

que hay una función denominada *getPastEvents*, esta función se encarga de buscar por toda la cadena de bloques el evento que le pasemos por parámetro, pudiendo definir entre que bloques queremos que busque si queremos agilizar el proceso, y devuelve toda la información grabada por el evento indicado. A continuación muestro un ejemplo de cómo queda la función que he desarrollado para recoger los eventos de pago de un ayuntamiento, sería similar para recoger los demás eventos definidos en la plataforma.

```
ethereumController.movimientoPagosAyuntamiento = function(addressAyuntamiento,callback){

console.log("\nrecogiendo eventos de pago...");
const input = fs.readFileSync('solidity/Ayuntamiento.sol');
const output = solc.compile(input.toString(), 1);
const abi = JSON.parse(output.contracts[':Ayuntamiento'].interface);

var contract = new web3.eth.Contract(abi,addressAyuntamiento);

var Event = contract.getPastEvents('pago', {
    fromBlock: 0,
    toBlock: 'latest'
}, function(error, events){ })

.then(function(events){
  console.log(events);
  var array = [];
      for(var i = 0; i < events.length ; i++){

        array.push(events[i].returnValues);

          if(array.length == events.length){

            return callback(array);
            }
    }

});

}
```

Obtener datos de un SmartContract es una de las funciones más sencillas puesto que al no realizar ningún cambio en la cadena de bloques no requiere de una transacción y con ello no requiere de coste de *gas*. Lo único que hay que hacer es llamar al método web3.eth.contract pasándole por parámetros el *ABI* y la dirección del SmartContract que localizará en la red el contrato y generará el objeto del SmartContract. Una vez tenemos el objeto podemos llamar a cualquier método definido dentro del SmartContract el cuál se encargue de obtener datos del contrato y nos devolverá los datos que hayamos definido en la función del contrato. Por ejemplo, como he mencionado antes, he desarrollado dentro del SmartContract Cliente.sol la función *getPersonalDataClient*, entre otras, para obtener los datos personales de un cliente. Ahora para recuperar esos datos desde node.js a través de web3, he desarrollado la siguiente función, que servirá la misma arquitectura para realizar cualquier otra función que requiera obtener datos de un SmartContract.

```
ethereumController.obtenerDatosCliente = function(addressCliente,callback){

console.log("\nrecogiendo contrato busqueda...");
const input = fs.readFileSync('solidity/Cliente.sol');
const output = solc.compile(input.toString(), 1);
const abi = JSON.parse(output.contracts[':Cliente'].interface);

var contract = new web3.eth.Contract(abi, addressCliente);

console.log("\nenviando call...");

contract.methods.getPersonalDataClient(addressCliente).call({from: '0x0b3ecA6DFFE4fFF33dAdC10DAE4cF8c4DE489857'})
.then(function(result){
    return callback(result);
});

}
```

Para modificar o añadir datos a un SmartContract si que se necesita realizar una transacción ya que estamos alterando la cadena de bloques, y algún "minero" tendrá que hacer constancia de esto y grabarlo en la cadena, con lo cual requiere un coste de *gas* y una cuenta responsable de esto. Por tanto tendremos que hacer uso de la función "send" para indicar qué cuenta es la responsable, y cantidad y precio de *gas* dispuestos a gastar. Por lo demás la metodología que sigue es similar a la de extraer datos, ya que hay que localizar primero el contrato y después llamar al metrodo definido dentro del SmartContract que es el encargado de realizar los cambios dentro del contrato. Una función por ejemplo que requiere cambiar el estado de un SmartContract es la función *pagarCompany* definida dentro del contrato Ayuntamiento.sol que se encarga de transferir Ether del contrato Ayuntamiento.sol al contrato Company.sol, y por tanto genera un cambio en el balance de ambos contratos y es por ello por lo que se necesita realizar una transacción para grabar esto en la cadena de bloques. A continuación muestro el desarrollo realizado para esta función que es similar para todos los casos en los que se requiera modificar datos de un SmartContract.

```
ethereumController.pagarCompany = function(pago,direc,name,callback){

console.log("\nrecogiendo contrato busqueda...");
const input = fs.readFileSync('solidity/Ayuntamiento.sol');
const output = solc.compile(input.toString(), 1);
const abi = JSON.parse(output.contracts[':Ayuntamiento'].interface);

var contract = new web3.eth.Contract(abi, pago.adrAyuntamiento);
console.log("\nenviando transacción...");
// using the promise
contract.methods.pagarCompany(pago.adrAyuntamiento,name,direc,pago.cantidad,pago.pass)

.send({
    from: "0x0b3ecA6DFFE4fFF33dAdC10DAE4cF8c4DE489857",
    jsonInterface : abi,
    gas: 100000,
    gasPrice: '10000',
}, function(error, transactionHash){

})
.on('error', function(error){
  console.log(error);
})
.on('transactionHash', function(transactionHash){
  console.log("\ntransactionHash:" + transactionHash);
console.log("\nRealizando pago...");
})
.on('receipt', function(receipt){

})
.on('confirmation', function(confirmationNumber, receipt){
 })
.then(function(newContractInstance){
    //console.log("newContractInstance:"+  newContractInstance.options.address) $
})

.then(function(result){
  console.log("es " + result);
    return callback(result);
});

}
```

5 Integración, pruebas y resultados

5.1 Pruebas de Smart Contracts en Remix

Como he mencionado en la parte de desarrollo, todos los Smart Contracts que he desarrollado para este TFG, los he ido probando en el IDE de Remix antes de integrarlos en mi red privada. El uso de Remix es bastante intuitivo y sencillo, permite escribir el código de contratos inteligentes, compilarlos e interactuar con ellos. La interfaz de usuario contiene en la parte izquierda el directorio de archivos que estas desarrollando, en la parte central superior la hoja de desarrollo de código, en la parte central inferior la consola del compilador, y en la parte derecha se encuentra la parte de compilador, ejecutable, configuración, análisis y depuración:

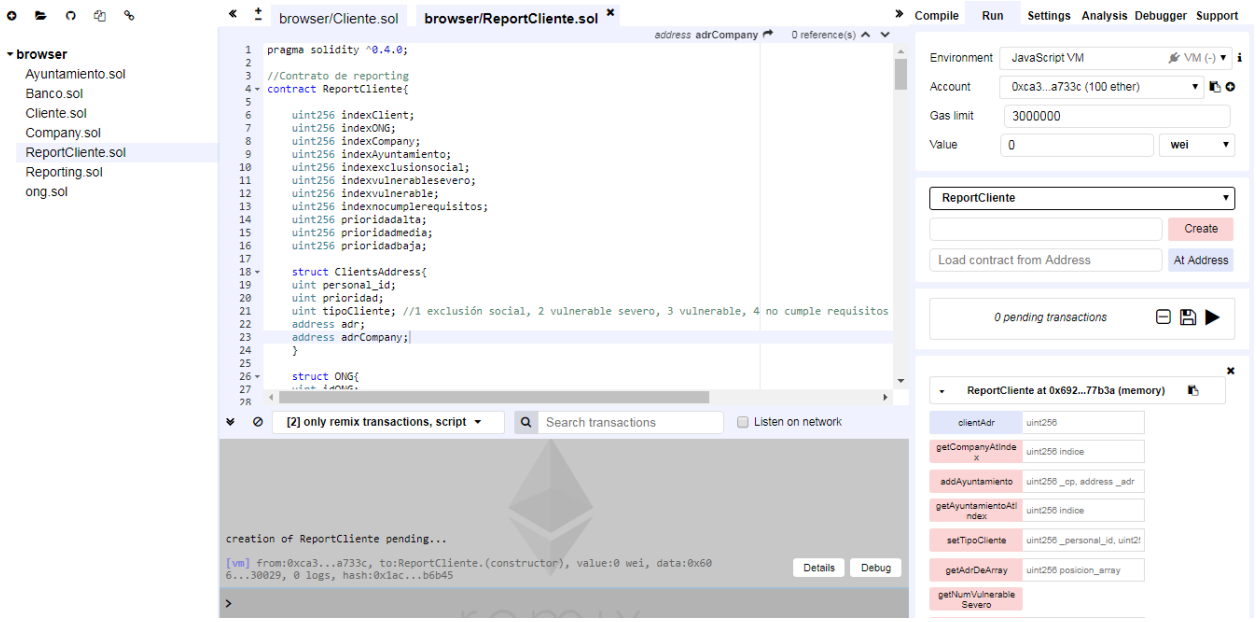


Figura 5-1: Interfaz de usuario de Remix

A continuación mostraré la metodología que he utilizado para probar los contratos, mostraré el caso de Cliente.sol que será suficiente ya que los demás Smart Contracts los he probado de la misma forma.

Una vez escrito el código del Smart Contract, lo primero que he hecho ha sido compilarlo en el apartado “Compile” para comprobar que no he tenido errores de código. Si hay algún error de código el compilador indica cuál es el error y en que punto se encuentra. En caso contrario la compilación se efectua con éxito y procedo a ejecutarlo. Para ejecutarlo he introducido los parámetros que requiere el constructor y pulsado en “Create”. Al ejecutarlo la consola devuelve toda la información de la transacción de la siguiente forma:

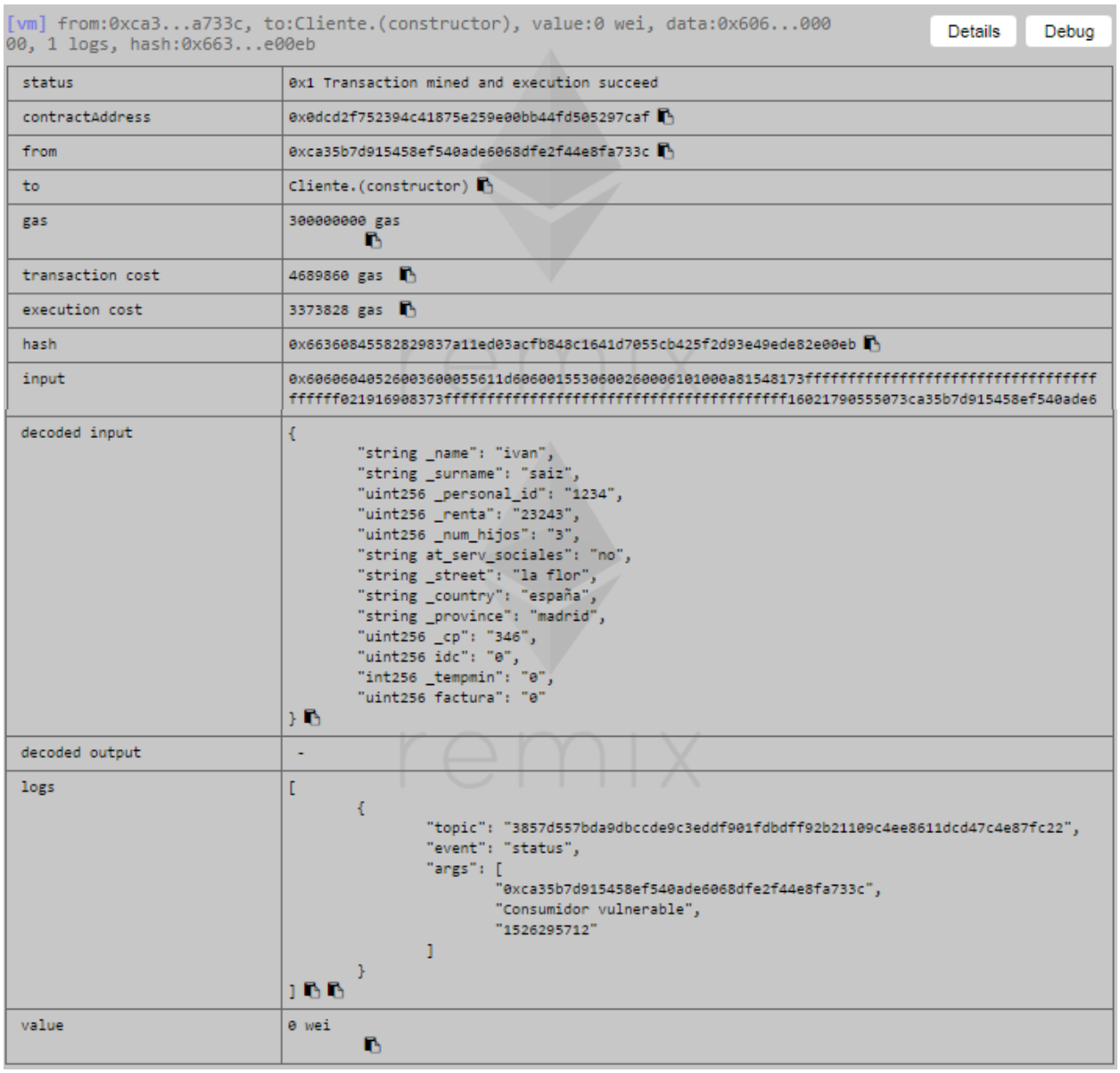


Figura 5-2: Datos de salida al desplegar un Smart Contract en Remix

La información mostrada por consola es muy útil ya que por ejemplo nos muestra el coste de la transacción, y con este dato a la hora de desplegar el contrato en mi red, puedo establecer el *gaslimit* superior al coste que me ha devuelto Remix y así evitar quedarme corto de *gas*. Vemos que muestra también la dirección hash del Smart Contract una vez desplegado, el hash de la transacción, la entrada de parámetros y los eventos grabados entre otros.

En la parte derecha de la pantalla se muestran todas las funciones que tiene el SmartContract y puedes interactuar con estas para comprobar su correcto funcionamiento. Por ejemplo para el caso de Cliente.sol tengo estas funciones:

Figura 5-3: Ejemplo de uso de funciones de un Smart Contract en Remix (1)

Para probar las funciones he tenido que introducir los parámetros que requiere la función y pulsar sobre el nombre de ésta, a continuación muestro el ejemplo de interactuar con la función clientDataHouse del Smart Contract Cliente.

Figura 5-4: Ejemplo de uso de funciones de un Smart Contract en Remix (2)

5.2 Integración y pruebas de la plataforma en red propia.

Tras las pruebas de los Smart Contract en Remix y el desarrollo completo de la plataforma integrada en la red que he desplegado para este TFG, he probado la creación de todos los Smart Contract en esta red que se generan interactuando con la aplicación web.

Para probar todos los Smart Contract desde la aplicación web hay que acceder a las distintas pantallas y rellenar los datos solicitados. En el **Anexo B**, se pueden ver todas las pantallas desde las cuales se capturan los datos que se introducen en la cadena de bloques. Una vez introducidos los datos, dependiendo la complejidad del Smart Contract y para la máquina utilizada en este trabajo (4GB RAM sin mucho poder computacional), una dificultad para la prueba de trabajo similar a la que se establece en la red pública de Ethereum (200000000*)* he comprobado que tarda una media de 25 segundos el nodo en resolver la prueba de trabajo y así introducir los Smart Contracts en la cadena de bloques. En la siguiente imagen muestro un ejemplo del registro que voy imprimiendo por consola (he invertido el color para evitar el negro) a la hora de desplegar un Smart Contract, para este caso el contrato cliente.sol:

```
Cliente {
  name: 'Iván',
  surname: 'Saiz',
  perID: 1234,
  renta: '27000',
  numHijos: 0,
  street: 'Calle la flor',
  country: 'España',
  province: 'Madrid',
  cp: '28100',
  atservsociales: 'NO',
  idc: '1234',
  factura: '50' }

desplegando contrato cliente...
es4712388

transactionHash:0x2da420807343fb81760d377e024520782e74e66589a2c3cbc9cc2710a882b0
cf

Recibiendo dirección contrato...
dirección contrato:0x01C82fBfe0212f3967678DC8a82535b3EBDe2828
contrato introducido
```

Figura 5-5: Datos de salida al desplegar un Smart Contract en red propia

Además del correcto funcionamiento de todos los Smart Contract, he probado también la realización de pagos, comprobado si graba correctamente los eventos de pago. Para esto a la hora de realizar un pago imprimo por consola toda la información de donde se ha grabado el evento, número de bloque, hash de la transacción, hash del bloque, *gas* utilizado, entre otros datos de información, obteniendo el siguiente resultado para uno de los pagos probados:

Figura 5-6: Prueba de pagos entre Smart Contracts a través de la plataforma

Una vez grabado el evento, he comprobado que puedo recoger estos eventos grabados. Para esto he ido al apartado “movimientos” de la plataforma desde donde llamo a la función encargada de recuperar todos los eventos de pago, y efectivamente los recupera bien como se puede ver en la siguiente imagen donde recupero el evento que se ha grabado en la anterior imagen.

```
recogiendo eventos de pago...
[ { address: '0x8b1F3A84b90893B056e7E304C111DF738e6f5298',
    blockNumber: 965428,
    transactionHash: '0x113f93decc494cbe0fec8df242e82490f86a355616785f765f0bf32409e16071',
    transactionIndex: 0,
    blockHash: '0xf96e7b582a777cb3b5f557765d84f0f5a828e5b9427c8ce2443ea6f2cff14cf9',
    logIndex: 1,
    removed: false,
    id: 'log_01d73b1c',
    returnValues:
     Result {
       '0': '0x7Df95E1d4B08ba08dbd21fFE84dB6C2e51eFc9b6',
       '1': 'Alcobendas',
       '2': '-',
       '3': '55',
       '4': '1526401184',
       '5': '95',
       adrAyuntamiento: '0x7Df95E1d4B08ba08dbd21fFE84dB6C2e51eFc9b6',
       name: 'Alcobendas',
       signo: '-',
       amount: '55',
       time: '1526401184',
       balance: '95' },
    event: 'pago',
    signature: '0xb32d6e3560e1bac73c445fdf6e68078ad1d54c860da4d0dcb949eb5b32d000d9',
    raw:
     { data: '0x0000000000000000000000007df95e1d4b08ba08dbd21ffe84db6c2e51efc9b60000000000
0000000000000000000000000000000000001000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000005f00000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000012d
       topics: [Array] } } ]
[ Result {
    '0': '0x7Df95E1d4B08ba08dbd21fFE84dB6C2e51eFc9b6',
    '1': 'Alcobendas',
    '2': '-',
    '3': '55',
    '4': '1526401184',
    '5': '95',
    adrAyuntamiento: '0x7Df95E1d4B08ba08dbd21fFE84dB6C2e51eFc9b6',
    name: 'Alcobendas',
    signo: '-',
    amount: '55',
    time: '2018-05-15, 6:19 PM',
    balance: '95' } ]
```

Figura 5-7: Prueba de recogida de eventos de pago

6 Conclusiones y trabajo futuro

6.1 Conclusiones

A lo largo del este TFG que he ido desarrollando siempre con el apoyo de mi tutor y mi ponente, a pesar de haber sido un trabajo duro y de mucha dedicación, hemos podido ir comprobando como cumplíamos con los objetivos propuestos. Desde el inicio con la investigación semana tras semana cada vez aprendía más y los conceptos quedaban más claros. La investigación no fue una tarea fácil puesto que se trata de una tecnología bastante reciente. Una vez comencé con el desarrollo gracias a las reuniones semanales que tenía con el tutor incluso diarias en ocasiones de picos de trabajo en los cuales veíamos avances diariamente, comenzamos a ver los resultados propuestos como objetivos, sintiendo gran satisfacción la primera vez que tras instalar Ethereum y desplegar los nodos conseguimos desarrollar el primer Smart Contract y desplegarlo en nuestra red, primer objetivo cumplido: desplegar una red privada Ethereum que permita utilizar el proyecto piloto a desarrollar sin los costes que requiere hacerlo en la red pública Ethereum. Gracias a reuniones que tuvimos también con una de las compañías eléctricas más importantes del país, pudimos ir definiendo mejor las funcionalidades y el diseño, además de ver el gran interés que muestran grandes compañías en esta tecnología, lo cual nos daba un mayor impulso y ganas de seguir trabajando y aprendiendo.

Destacar que el IDE de Remix me ha sido bastante útil a la hora de desarrollar Smart Contracts con Solidity, lo recomiendo mucho para aquellos que quieran comenzar a aprender y a desarrollar contratos inteligentes, ya que te permite compilarlos y probar toda su funcionalidad. Tras el desarrollo completo de todos los Smart Contracts que cumplían ya el resto de objetivos: grabar un registro único de todos los solicitantes del bono social en la cadena de bloques de la red anteriormente desplegada, automatizar la concesión del bono social a través de un Smart Contract, desarrollar una función que permita pagar las facturas de los más vulnerables, dejando registro único de estos pagos y la posibilidad de realizar donaciones para este fin, dejando toda la trazabilidad en la cadena de bloques, llegó la hora de integrarlos con la plataforma web y establecer la comunicación a través de web3js para así conseguir que la plataforma web en conjunto cumpliese todos los requisitos. Gracias a la gran documentación que tiene la web oficial de web3js rápidamente aprendí a utilizar esta API, consiguiendo otro de los objetivos fundamentales que era integrar los Smart Contracts con la aplicación web. Tras toda la integración, realicé numerosas pruebas de la aplicación completa comprobando que habíamos cumplido con el objetivo del TFG.

Después de estos meses de trabajo y la finalización de este proyecto he llegado a la conclusión de que es posible realizar infinitas aplicaciones descentralizadas para casos de usos reales funcionando en una cadena de bloques, consiguiendo tener trazabilidad absoluta de todas las operaciones y datos.

Por último destacar que viendo el coste y tiempo de ejecución de transacciones veo que lo más conveniente es utilizar una red híbrida que tenga un coste de *gas* nulo y utilice un algoritmo de consenso que disminuya el tiempo de ejecución de las transacciones, pero que a su vez esté constituida por miembros sin intereses comunes. Investigando en el mercado, una posible red para conseguir esto sería Alastria.

6.2 Trabajo futuro

Como trabajo futuro se propone avanzar con la plataforma integrando servicios para la verificación de datos con distintas compañías eléctricas, con el MINETAD y con los

servicios sociales, para acercarse más al producto real pudiendo llegar a implantarse esta plataforma para uso real por parte de todas las compañías eléctricas y la administración.

Otra línea de trabajo futuro que se propone es la realización de un aplicación móvil que permita a los servicios sociales marcar a los solicitantes en riesgo de exclusión social una vez que realizan la visita a la familia, pudiendo así agilizar el proceso de clasificación de clientes. La aplicación móvil que se propone como trabajo futuro podría también permitir realizar el pago de las facturas con un solo botón. Esta aplicación debería interactuar con los Smart Contracts creados en este TFG.

Una propuesta más es la integración de esta plataforma en una red híbrida, por ejemplo Alastria, para probar el rendimiento de la plataforma en una red de este tipo que sería la ideal para llegar a utilizar esta herramienta por parte de compañías eléctricas y administración pública.

Referencias

[1] Documentación oficial Ethereum. *https://www.ethereum.org/*

[2] SHA256. *https://es.wikipedia.org/wiki/SHA-2*

[3] Vitalik Buterin. *"On public and private Blockchains"* *https://www.coindesk.com/vitalik-buterin-on-public-and-private-blockchains/*

[4] *https://github.com/ethereum/go-ethereum/wiki/Installation-Instructions-for-Ubuntu*

[5] *https://github.com/ethereum/go-ethereum/wiki/Setting-up-private-network-or-local-cluster*

[6] *https://github.com/ethereum/go-ethereum/wiki/Private-network*

[7] *https://github.com/ethereum/go-ethereum/wiki/Command-Line-Options*

[8] *https://github.com/ethereum/go-ethereum/wiki/Connecting-to-the-network*

[9] *https://github.com/ethereum/wiki/wiki/JavaScript-API*

[10] *https://solidity.readthedocs.io/en/v0.4.23*

[11] Proof of Burn. *http://slimco.in/proof-of-burn-eli5-es*

[12] Documentación oficial Solidity. *https://solidity.readthedocs.io/en/v0.4.23*

[13] https://www.boe.es/buscar/doc.php?id=BOE-A-2017-11505

[14] EC brands. *"Arranca el plan contra la pobreza energética en España"* *https://brands.elconfidencial.com/sociedad/2017-07-24/plan-pobreza-energetica-espana-gas-natural_1418471*

[15] Documento BOE-A-2018-4750. *"Bono social de luz"* *https://www.boe.es/diario_boe/txt.php?id=BOE-A-2018-4750*

[16] Documentación completa web3. *https://github.com/ethereum/wiki/wiki/JavaScript-API#web3versionapi*

[17] Comisión Nacional de los Mercados y Competencia. "*Bono social"* *https://www.cnmc.es/bono-social*

Glosario

API	Application Programming Interface
DApp	Decentralized Application
P2P	Peer-to-Peer
ABI	Application Binary Interface
EVM	Ethereum Virtual Machine
IDE	Integrated Development Environment
PVPC	Precio Voluntario para el Pequeño Consumidor
DNI	Documento Nacional de Identidad
IPREM	Indicador Público de Renta de Efectos Múltiples

Anexos

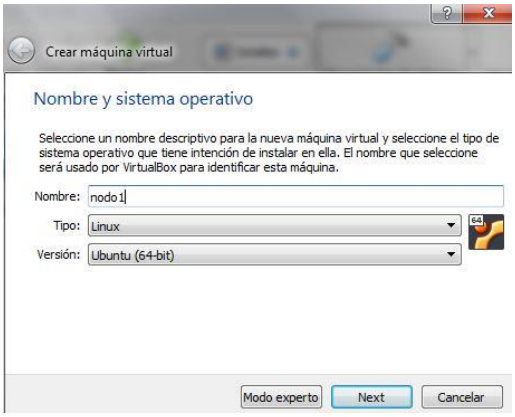
A Manual de instalación y despliegue de red privada Ethereum

Este anexo contiene notas y explicaciones sobre la instalación, configuración y credenciales del despliegue de dos nodos en una red privada Ethereum. Se verá que la parametrización elegida es un ejemplo funcional pero hay multitud de opciones y variables que se pueden configurar según nuestras necesidades.

Instalación de Ubuntu Linux 16.04 LTS

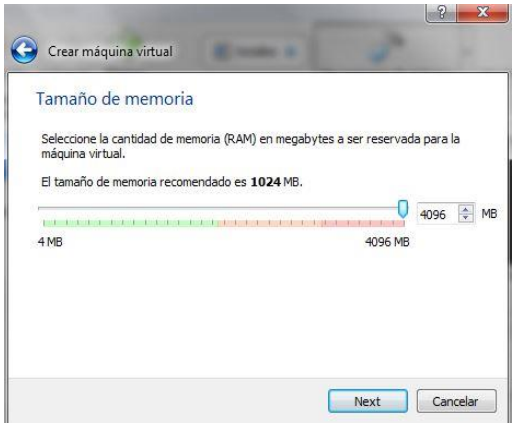
La red la he desplegado en dos servidores virtuales (4GB de RAM y 50GB de disco duro para cada uno) con Ubuntu Linux 16.04 LTS, cada uno de los servidores es un nodo de la red. Ambos servidores se encuentran en un portátil HP840G. Para esto lo primero que se ha hecho ha sido descargar VirtualBox 5.2.4 de Oracle e instalar dos máquinas virtuales con dichas características. Los pasos a seguir para la instalación son los siguientes:

- El primer paso es darle un nombre a la máquina y elegir el sistema operativo y la versión, en mi caso se llama nodo1 con sistema operativo Linux y versión Ubuntu.



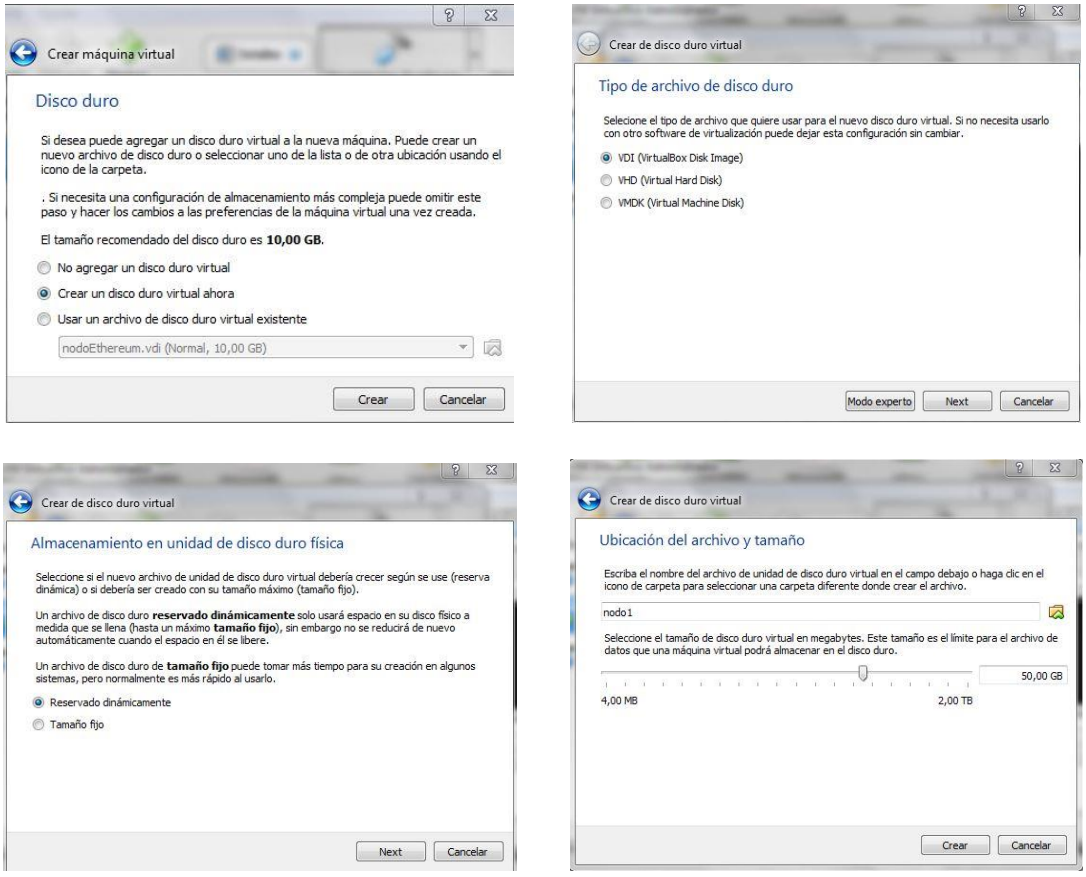
Instalación Ubuntu Linux en MV (1)

- El siguiente paso es escoger el tamaño de memoria RAM que deseamos en nuestra máquina, en mi caso 4GB, no es recomendable crear una red privada con Ethereum con poca memoria RAM ya que como hemos visto antes el "minado", es decir, la prueba de trabajo a la hora de incluir bloques en nuestra blockchain requiere un elevado coste computacional y con poca memoria RAM el sistema se vería muy relentizado.



Instalación Ubuntu Linux en MV (2)

- A continuación se crea el disco duro virtual, eligiendo: tipo de archivo de disco duro VDI (VirtualBox Disk Image), Reservado dinámicamente y el tamaño deseado, en mi caso 50GB. Con estos sencillos pasos tendríamos ya preparada una máquina virtual para instalar Ethereum y montar la red blockchain.



Instalación Ubuntu Linux en MV (3)

Instalación de software Ethereum y despliegue de dos nodos

Una vez creadas ambas máquinas virtuales agregamos el repositorio de software Ethereum, actualizamos los repositorios e instalamos el software. Los comandos a ejecutar son los siguientes:

Sudo add-apt-repository –y ppa:ethereum/ethereum (agrega el repositorio de software Ethereum)

Sudo apt-get update (actualiza los repositorios)

Sudo apt-get install ethereum (instala el software de Ethereum)

Con estos tres comandos tenemos Ethereum instalado en las máquinas y con ello *Geth*, Geth es el cliente oficial de Ethereum escrito en el lenguaje de programación Go de Google, se puede describir como la interfaz de línea de comandos que nos permitirá interactuar con nuestros nodos de la red. Se puede utilizar Geth de tres formas distintas:

- Consola JavaScript: geth se puede iniciar con una consola interactiva, que proporciona un entorno de tiempo de ejecución de JavaScript que expone una API de JavaScript para interactuar con un nodo de la red. JavaScript Console API incluye la API de web3js, así como una API de administración adicional.
- Servidor JSON-RPC: geth se puede iniciar con un servidor json-rpc que expone la API JSON-RPC
- Se puede utilizar directamente con comandos, en el anexo "Comandos para Geth" se detallan los parámetros de la línea de comando y los subcomandos.

El siguiente paso es crear las carpetas y ficheros necesarios para almacenar el blockchain, que son los siguientes:

/etc/ethereum => Donde se almacenan los archivos de configuración.
/var/lib/ethereum => Donde se almacena el datastore de Geth.
/var/log/ethereum => Donde se almacenan los logs del servidor Geth.

A continuación se crea el archivo de configuración de nuestra red privada en formato .json, que debe llamarse *genesis.json*, y ubicarse en el directorio /etc/ethereum. Este archivo contiene toda la información del bloque génesis, el bloque génesis es el bloque con el que comienza cualquier blockchain y permite establecer los parámetros de configuración con las características que tendrá nuestra blockchain. Este archivo contiene distintos parámetros de configuración:

- config: La configuración del blockchain
- chainId: Es el identificador único de la cadena de bloques, será importante este dato a la hora de añadir nodos a esta cadena porque haremos referencia a ella a través de este identificador.
- homesteadBlock: Número de bloque con el que comenzará la cadena.
- epi155Block: epi significa Propuesta de Mejora Ethereum, donde los desarrolladores proponen ideas sobre cómo mejorar Ethereum y contribuir a este proyecto.
- difficulty: Determina la dificultad de minería. Para una red privada de pruebas se aconseja establecer este valor bajo para que no tenga que esperar demasiado tiempo a la hora de minar bloques.
- gasLimit: Determina el límite del costo del gas por bloque. Es aconsejable establecer este valor alto para evitar ser limitado cuando se realizan transacciones que tienen un alto coste.

- alloc: dirección prefinanciada, el primer parámetro de cada uno es la dirección. Necesita ser una cadena hexadecimal de 40 dígitos (160 bit, un dígito hexadecimal es 4 bit). Esto no crea una cuenta sino que a una cuenta ya existente se le otorga el número de Ether dentro de nuestra red que se establece en el parámetro "balance".

El archivo *genesis.json* que he definido para nuestra cadena de bloques es el siguiente:

```
{
   "config": {
      "chainId": 15, //Identificador de la blockchain
      "homesteadBlock": 0, //Número de bloque con el que comienza
      "eip155Block": 0, //sin propuestas
      "eip158Block": 0
   },
   "difficulty": "200000000", //Dificultad de minería
   "gasLimit": "2100000", //Límite de coste de gas
   "alloc": {
      "7df9a875a174b3bc565e6424a0050ebc1b2d1d82": { "balance": "300000" },
      "f41c74c9ae680c1aa78f42e5647a62f353b7bdde": { "balance": "400000" }
   } //cuentas con su balance correspondiente
}
```

Una vez generado el archivo genesis.json se inicializa el bloque utilizando Geth con el siguiente comando:

geth –datadir "/var/lib/ethereum/" init /etc/ethereum/genesis.json

Para añadir un nodo a la red se hace también con geth. A la hora de inicializar el nodo se puede definir varios parámetros:

- Networkid: sirve para indicar a que red quiero añadir el nodo, en mi caso el valor es 15.
- Verbosity: nivel de detalle del log, las distintas opciones son 0=silent, 1=error, 2=warn, 3=info, 4=debug, 5=detail, en mi caso he elegido el valor por defecto que es 3.
- Rpc: servicio de HTTP-RPC.
- Rpcaddr: Dirección donde va a estar el servidor rpc, yo he elegido 0.0.0.0 lo que equivale a localhost, la dirección ip propia de la máquina.
- Rpcapi: Este parámetro sirve para indicar las distintas API de rpc que se van a utilizar. Es recomendable utilizar: admin: para administrar el nodo Geth, debug: para depurar el nodo Geth, miner: permite gestionar los parámetros de minería y generar el DAG (Directed Acyclic Graph), personal: Para la administración de cuentas, txpool: API le da acceso a varios métodos RPC no estándar para inspeccionar los contenidos del grupo de transacciones que contiene todas las transacciones actualmente pendientes, así como las que están en cola para el procesamiento futuro, web3: API para establecer comunicación desde una aplicación desarrollada en JavaScript con la red Ethereum.
- Rpcport: Sirve para indicar el puerto de comunicación de rpc, por defecto si no se indica otro distinto este puerto es el 8545, pero se puede establecer otro distinto, en mi caso he mantenido el 8545.

El comando final que he utilizado para poner en marcha el primer nodo con los ya mencionados parámetros es el siguiente:

nohup geth --datadir="/var/lib/ethereum/" --networkid="15" --verbosity 3 --rpc --rpcaddr "0.0.0.0" --rpcapi admin,eth,miner,web3,personal.txpool,net,debug –rpcport 8545 2>> /var/log/ethereum/ethereum.log &

Para el segundo he realizado los mismos pasos que para el primero, pero en otra máquina virtual con las mismas características, sólo hay una diferencia que es la siguiente: hay que añadir los nuevos nodos que se quieran añadir a la instancia principal, es decir, al primer nodo que se crea. Para hacer esto se añade el parámetro bootnodes en el que se indica la ruta del enode principal, la ruta del enode se obtiene de los logs del nodo principal y tiene el formato pubkey1@ip:port. Por tanto el comando utilizado para el segundo nodo es el siguiente (por seguridad no muestro la ip y el puerto del enode):

nohup geth --datadir="/var/lib/ethereum/" --networkid="15" --verbosity 3 --rpc --rpcaddr "0.0.0.0" --rpcapi admin,eth,miner,web3,personal.txpool,net,debug –rpcport 8545 " --bootnodes "enode://e9f66dda3064e670b6d2344683b1d80bbe872da1801aa2fad04184180553006 56cf12ea72db17aa599b99fc9bcef423f6aa46b955c85f5a2c4e06344fd06b9ac@ip:port 2>> /var/log/ethereum/ethereum.log &

Ya estaría creada la red privada Ethereum con dos nodos, para conectarse a estos nodos se puede hacer de dos formas distintas:

- Desde cliente Ethereum con geth con el comando:

  geth attach http://localhost:8545

  Que abre la siguiente consola:

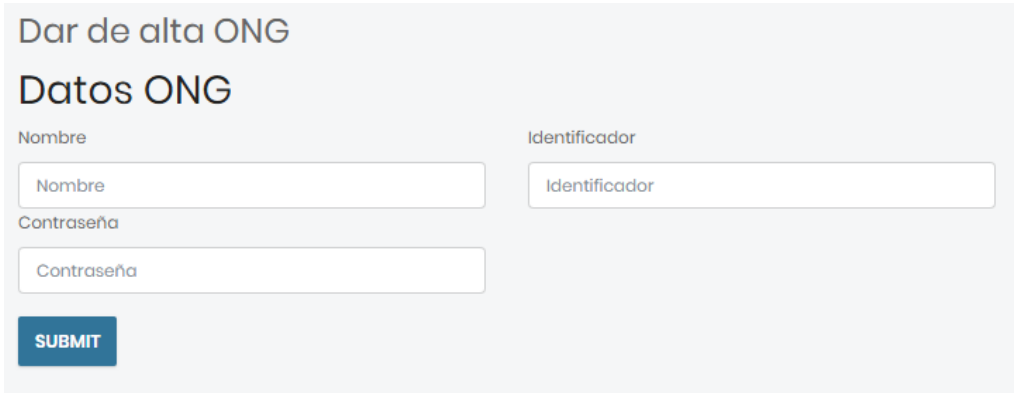
```
instance: Geth/v1.8.2-stable-b8b9f7f4/linux-amd64/go1.9.4
coinbase: 0x2bcc06bdb37ac65651e51e678fa0392c429aef5b
at block: 838 (Thu, 26 Apr 2018 19:11:08 CEST)
 datadir: /var/lib/ethereum
 modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txp
ool:1.0 web3:1.0

>
```

Consola nodo Geth

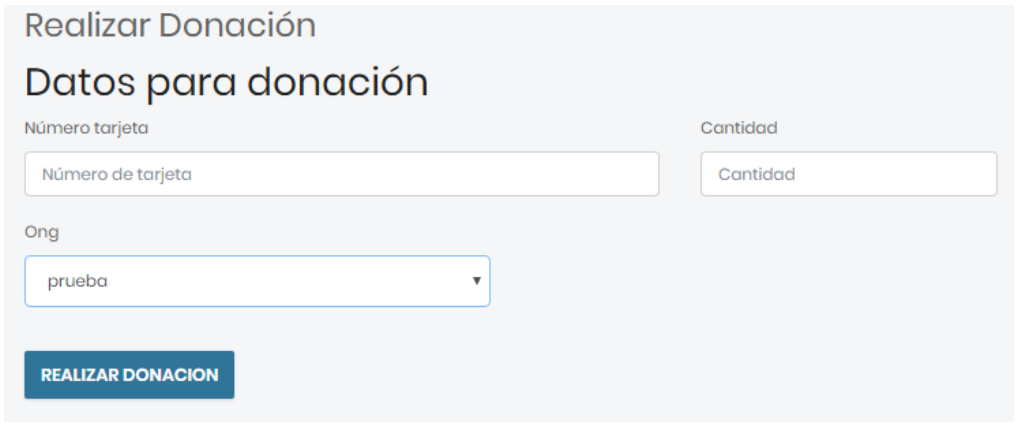
- Desde Javascript => Web3.providers.HttpProvider("http://localhost:8545")

B Pantallas de la plataforma web

Pantalla principal



Alta ONG

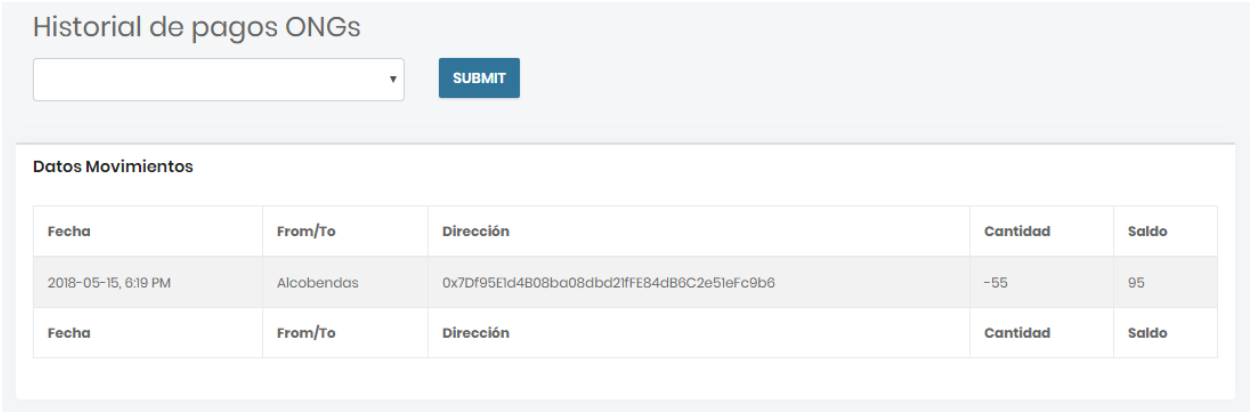


Pantalla donaciones

Alta Ayuntamientos

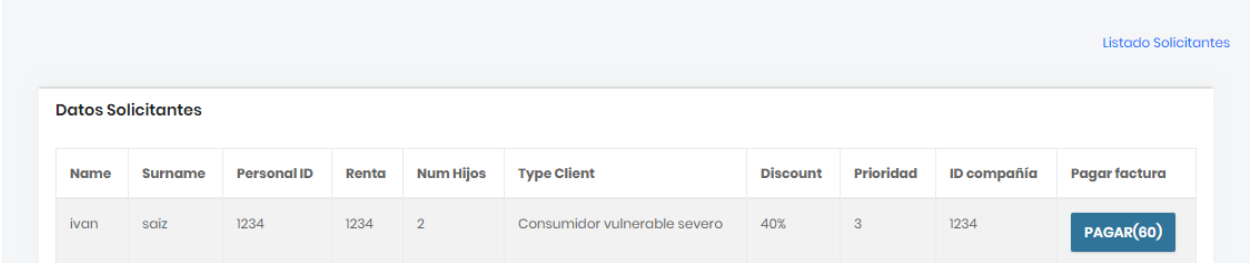
Alta Compañías

Pago a compañías



Historial de pagos

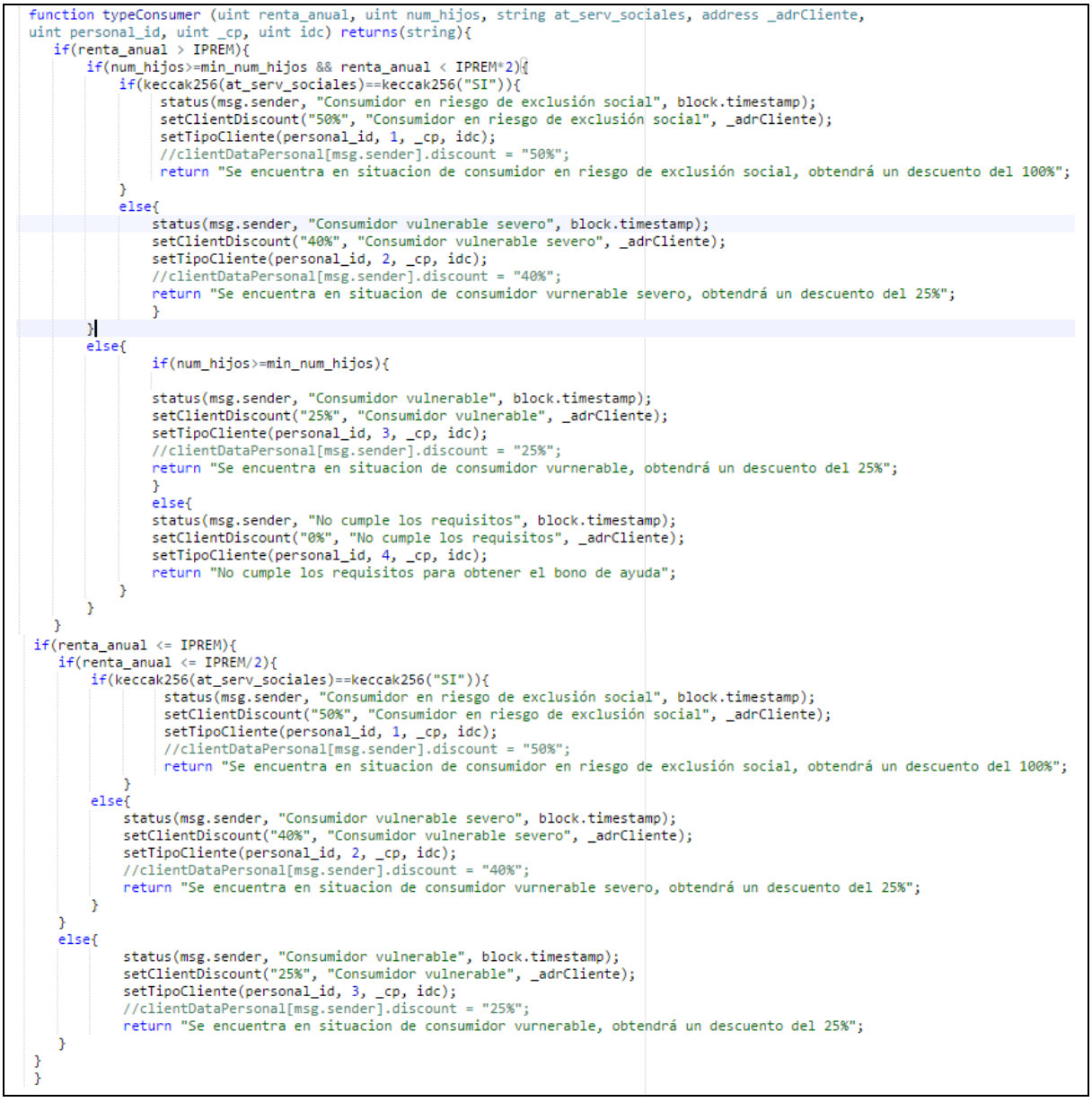
Pantalla solicitante



Listado solicitantes

C Partes importantes de código de los Smart Contracts

Función typeConsumer: Contiene las cláusulas que determinar qué tipo de cliente eres en función de los parámetros solicitados por pantalla.



```
function typeConsumer (uint renta_anual, uint num_hijos, string at_serv_sociales, address _adrCliente,
uint personal_id, uint _cp, uint idc) returns(string){
   if(renta_anual > IPREM){
       if(num_hijos>=min_num_hijos && renta_anual < IPREM*2){
           if(keccak256(at_serv_sociales)==keccak256("SI")){
                status(msg.sender, "Consumidor en riesgo de exclusión social", block.timestamp);
                setClientDiscount("50%", "Consumidor en riesgo de exclusión social", _adrCliente);
                setTipoCliente(personal_id, 1, _cp, idc);
                //clientDataPersonal[msg.sender].discount = "50%";
                return "Se encuentra en situacion de consumidor en riesgo de exclusión social, obtendrá un descuento del 100%";
           }
           else{
               status(msg.sender, "Consumidor vulnerable severo", block.timestamp);
               setClientDiscount("40%", "Consumidor vulnerable severo", _adrCliente);
               setTipoCliente(personal_id, 2, _cp, idc);
               //clientDataPersonal[msg.sender].discount = "40%";
               return "Se encuentra en situacion de consumidor vurnerable severo, obtendrá un descuento del 25%";
               }
       }
       else{
               if(num_hijos>=min_num_hijos){

               status(msg.sender, "Consumidor vulnerable", block.timestamp);
               setClientDiscount("25%", "Consumidor vulnerable", _adrCliente);
               setTipoCliente(personal_id, 3, _cp, idc);
               //clientDataPersonal[msg.sender].discount = "25%";
               return "Se encuentra en situacion de consumidor vurnerable, obtendrá un descuento del 25%";
               }
               else{
               status(msg.sender, "No cumple los requisitos", block.timestamp);
               setClientDiscount("0%", "No cumple los requisitos", _adrCliente);
               setTipoCliente(personal_id, 4, _cp, idc);
               return "No cumple los requisitos para obtener el bono de ayuda";
           }
       }
   }
if(renta_anual <= IPREM){
   if(renta_anual <= IPREM/2){
       if(keccak256(at_serv_sociales)==keccak256("SI")){
                status(msg.sender, "Consumidor en riesgo de exclusión social", block.timestamp);
                setClientDiscount("50%", "Consumidor en riesgo de exclusión social", _adrCliente);
                setTipoCliente(personal_id, 1, _cp, idc);
                //clientDataPersonal[msg.sender].discount = "50%";
                return "Se encuentra en situacion de consumidor en riesgo de exclusión social, obtendrá un descuento del 100%";
           }
       else{
           status(msg.sender, "Consumidor vulnerable severo", block.timestamp);
           setClientDiscount("40%", "Consumidor vulnerable severo", _adrCliente);
           setTipoCliente(personal_id, 2, _cp, idc);
           //clientDataPersonal[msg.sender].discount = "40%";
           return "Se encuentra en situacion de consumidor vurnerable severo, obtendrá un descuento del 25%";
       }
   }
   else{
           status(msg.sender, "Consumidor vulnerable", block.timestamp);
           setClientDiscount("25%", "Consumidor vulnerable", _adrCliente);
           setTipoCliente(personal_id, 3, _cp, idc);
           //clientDataPersonal[msg.sender].discount = "25%";
           return "Se encuentra en situacion de consumidor vurnerable, obtendrá un descuento del 25%";
   }
}
}
```

Ayuntamiento.sol:

Recuerdo que siempre que programamos un Smart Contract, lo primero hay que indicar la versión, inicializar el contrato y definir la estructura de datos. Los datos que tiene que almacenar este Smart Contract son: nombre, C.P, dirección hash del contrato, contraseña, temperatura mínima y número de solicitantes diferenciados por el tipo de cliente.

```
pragma solidity ^0.4.0;

contract Ayuntamiento{

    //Estructura que contiene las variables con la información de cada ayuntamiento
    struct ayuntamientodata{
    string nombre;
    uint cpAyunta;
    address adr;
    string password;
    int tempmin;
    uint numExclusionSocial;
    uint numVulnerable;
    uint numSevero;
    uint numNoCumple;
    }
```

Como he mencionado en el apartado de diseño, el Smart Contract que se desplegará cada vez que se dé de alta en el sistema un ayuntamiento, deberá poder recibir donaciones de ONGS a la dirección de este contrato y realizar pagos a compañías. Para que un Smart Contract pueda recibir y enviar *Ether*, hay que indicarlo en el constructor, para indicarlo hay que definir el constructor como tipo *payable*.

```
function Ayuntamiento(uint _cp, string _nombre, int temp, string pass) payable {
//guarda la información del ayuntamiento
addAyunt(_nombre,_cp,temp,pass);
//guarda el código postal y la dirección del smartcontract  en el contrato maestro
addAyuntamiento(_cp,address(this));

}
```

Además de indicar en el constructor que este contrato es tipo *payable*, hay que añadir la función *payable* en el Smart Contract.

```
//Esta función se debe incluir en todos los SmartContract
//que vayan a recibir y enviar ether
  function() payable { }
```

De nuevo he establecido comunicación entre Ayuntamiento.sol y ReportCliente.sol para guardar en el contrato "maestro" la dirección del Smart Contract junto con el código postal.

```
ReportCliente reports =  ReportCliente (0xB7C08d63D07D05A0e39a60875bf8B475F840902c);
//Llamada a una función que esta dentro de ReportCliente
function addAyuntamiento(uint _cp, address _adr) public {
    reports.addAyuntamiento(_cp, _adr);
}
```

```
//Contrato de reporting
contract ReportCliente{
function ReportCliente() public{

}
    uint256 indexAyuntamiento;

    struct Ayunta{
    uint cp;
    address adrAyuntamiento;
    }

    mapping (uint => Ayunta) private ayuntaadr;

    address[] numAyuntamientos;

    function addAyuntamiento(uint _cp, address _adr){

    }

    }
```

Para pagar a una compañía eléctrica he definido la función *pagarCompany*, esta función también debe ser tipo *payable* ya que se va a encargar de enviar Ether al Smart Contract de una compañía. Para realizar un pago, solicito la contraseña del ayuntamiento y compruebo que es correcta, si es correcta, con la función *transfer* realizo la transferencia de Ether a la dirección del contrato de la compañía. Si se realiza el pago, lo grabo en un evento llamado *pago* para después desde node.js poder recuperar todo el registro de pagos. En este evento grabo la dirección de la compañía a la que se ha realizado el pago, el nombre de la compañía, el signo negativo puesto que es un pago y no una donación y por tanto disminuirá el balance del ayuntamiento, la cantidad transferida, la fecha y hora, y el balance con el que se queda el ayuntamiento después del pago.

```
//Esta función sirve para pagar ether a una compañía eléctrica,
//transfiere ether de un smartcontract a otro. Se pide una password para que tengan
//seguridad los pagos
  function pagarCompany(address adrAyuntamiento, string namecompany,
  address adrCompany, uint amount, string password) payable external returns(string) {
     if(keccak256(password)==keccak256(ayuntamientodatos[adrAyuntamiento].password)){

         adrCompany.transfer(amount);
         //Llamamos al evento pago para grabar el pago en blockchain
         pago(adrCompany, namecompany, "-", amount, block.timestamp, this.balance);

         return("Pago realizado con exito");
        }
     else{
         return("Contraseña incorrecta");
        }
    }
event pago(address adr, string name,
         string signo, uint amount, uint256 time, uint balance);
```

Uno de las complicaciones que me he encontrado al recuperar los eventos de pago, es que para recuperar estos eventos debes recuperarlos haciendo referencia a la dirección del Smart Contract donde se ha grabado el evento, y por tanto cuando quiero recuperar los pagos realizados de un ayuntamiento y las donaciones recibidas y mostrar el balance completo, los eventos estan grabados en distintos Smart Contracts, es decir, los pagos que

realiza un ayuntamiento están grabados en el Smart Contract Ayuntamiento.sol y las donaciones que hace una ONG a un ayuntamiento estan grabadas en el Smart Contract Ong.sol. Para solucionar este problema he desarrollado una función llamada *registrarPago* que es invocada desde node.js cuando una ONG realiza una donación a un Ayuntamiento y así tengo todos los movimientos de un ayuntamiento grabados en un mismo Smart Contract y ahora obtener todo el registro de pagos y balance es más sencillo. En el caso de esta función el signo es positivo puesto que supone un aumento en el balance.

```
function registrarPago(address adrOng, string nameOng, uint amount){
    pago(adrOng, nameOng, "+", amount, block.timestamp, this.balance);
}
```

Company.sol:

Siguiendo el mismo esquema anteriormente mencionado, lo primero indico cual es la versión utilizada. Para el caso de las compañías no necesito variables predefinidas ya que este Smart Contract me servirá para almacenar datos de la compañía, recibir pagos de facturas y obtener el balance de la misma. La información que almaceno es: nombre, identificador, dirección, contraseña, y número de solicitantes diferenciados por el tipo de cliente. Necesito definir la estructura de datos donde almaceno la información.

```
pragma solidity ^0.4.0;
contract Company{

    struct companydata{
    string nombre;
    uint id;
    address adr;
    string pass;
    uint numExclusionSocial;
    uint numVulnerable;
    uint numSevero;
    uint numNoCumple;
    }
```

Este Smart Contract que se desplegará cada vez que se dé de alta en el sistema una compañía eléctrica, deberá poder recibir pagos de ayuntamientos a la dirección de este contrato. Al igual que el anterior Smart Contract en este también hay que definir el constructor como tipo *payable,* y añadir la función *payable.*

```
function Company(uint _id, string _nombre, string pass) payable {
//guarda la información de la compañía
addComp(_nombre,_id,pass);
//guarda el identificador y la dirección del smartcontract en el contrato maestro
addCompany(_id,address(this));
}

//Esta función se debe incluir en todos los SmartContract que vayan a recibir ether
function() payable { }
```

Siguiendo el mismo diseño para todos los Smart Contracts que he desarrollado, establezco comunicación entre Company.sol y ReportCliente.sol, almacenando la dirección junto con el identificador.

```
ReportCliente reports =  ReportCliente (0xB7C08d63D07D05A0e39a60875bf8B475F840902c);

function addCompany(uint _id, address _adr) public {
reports.addCompany(_id, _adr);
}
//Contrato de reporting
contract ReportCliente{

function ReportCliente() public{

}
    uint256 indexCompany;

    struct Comp {

    address adrCompany;
    uint idCompany;

    }

    mapping (uint => Comp) private companyAdrs;

    address[] numCompany;

    function addCompany(uint _id, address _adr){

    }

    }
```

Para poder recuperar el registro de pagos recibidos por parte de los ayuntamientos, he seguido la misma estrategia que en Ayuntamiento.sol, he desarrollado la misma función *registrarPago* que contiene el evento *pago* y que es invocada desde node.js cuando un ayuntamiento paga una factura a la compañía, esto lo he hecho para poder recuperar luego todo el registro haciendo referencia a la dirección de este Smart Contract.

```
function registrarPago(address adrAyuntamiento, string nameAyuntamiento, uint amount){
    pago(adrAyuntamiento, nameAyuntamiento, "+", amount, block.timestamp, this.balance);
  }
event pago(address adrAyuntamiento, string name,
        string signo, uint amount, uint256 time, uint balance);
```

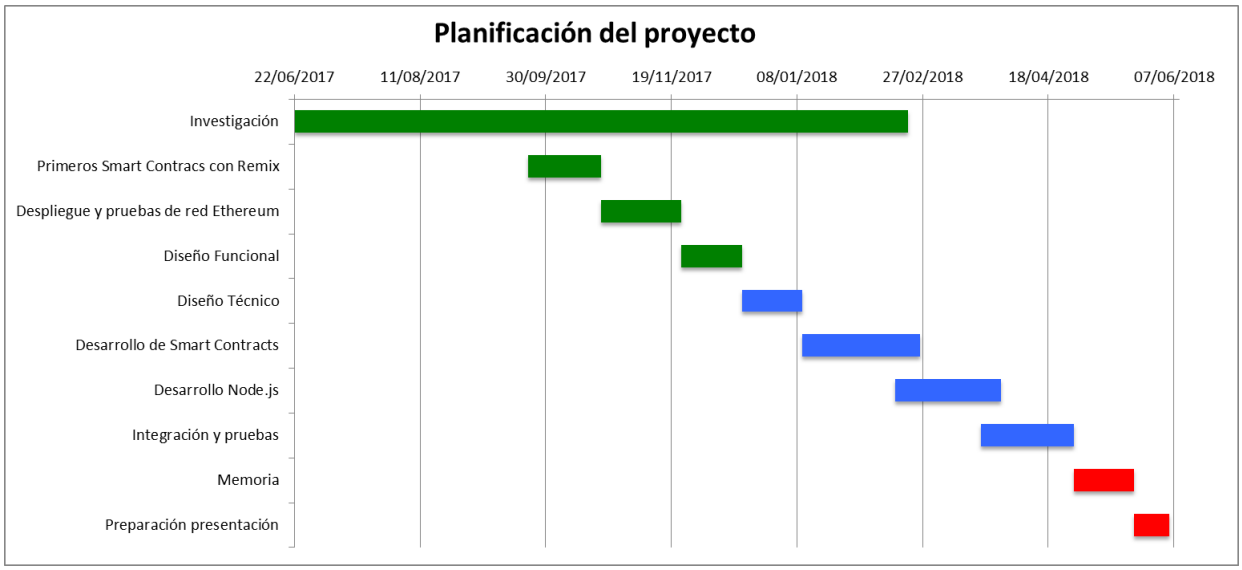
Ong.sol:

Por no repetirme no voy a explicar el desarrollo completo de este Smart Contract ya que es idéntico al de las compañías y ayuntamientos. La única novedad que incluyo en este Smart Contract es una función que he desarrollado para poder hacer donaciones desde cualquier cuenta de Ethereum, sin ser necesariamente una dirección de un Smart Contract desde la cual se transfiera el dinero. A esta función la he llamado *donacionONG*, ahora el parámetro que paso a la función transfer es *this.balance* que es el valor en Ether que se envía desde una cuenta Ethereum y lo envío a la dirección de la ONG (*adrOng)* que es la dirección propia del Smart Contract.

```
//Función para insertar dinero en el smartcontract ONG
//desde una cuenta de ethereum y no desde un smartcontract
 function donacionONG(address adrONG, address adrDonante, uint amount)
                    payable returns(bool) {

    adrONG.transfer(this.balance);
    pago(adrDonante, "Donante", "+", amount, block.timestamp, this.balance);
    return(true);

    }
```

D Planificación y metodología de trabajo

Nombre de la tarea	Fecha de inicio	Fecha final	Duración (dias)
Investigación	22/06/2017	21/02/2018	244
Primeros Smart Contracs con Remix	23/09/2017	22/10/2017	29
Despliegue y pruebas de red Ethereum	22/10/2017	23/11/2017	32
Diseño Funcional	23/11/2017	17/12/2017	24
Diseño Técnico	17/12/2017	10/01/2018	24
Desarrollo de Smart Contracts	10/01/2018	26/02/2018	47
Desarrollo Node.js	16/02/2018	30/03/2018	42
Integración y pruebas	22/03/2018	28/04/2018	37
Memoria	28/04/2018	22/05/2018	24
Preparación presentación	22/05/2018	05/06/2018	14



La metodología de trabajo que se ha seguido para la realización de este TFG ha sido Agile, esta metodología permite el desarrollo de proyectos que precisan de adaptación al cambio. El proyecto como se puede ver en la planificación se estructuro en distintas piezas que procuramos que no se alargasen más de un mes pudiendo ver resultados semanales en las tareas que iba desarrollando. Semanalmente tenía reuniones con mi tutor incluso en

ocasiones varias veces por semana, se trataban de reuniones breves pero muy importantes para ir viendo resultados y definiendo nuevas tareas para cumplir todos los objetivos a un ritmo constante. Puedo casi afirmar que la investigación ha sido una tarea que he realizado en casi la totalidad el proyecto, al tratarse de una tecnología reciente con constantes actualizaciones, he tenido que ir adaptando el proyecto a estas actualizaciones que cada vez eran más maduras.
También han sido muy útiles las reuniones y presentaciones de producto que hemos ido realizando con una de las compañías eléctricas más importantes del país, lo que nos ha facilitado adaptar el proyecto y las funcionalidades.